AFIT/GE-93D-10

DISCOVERY LEARNING IN AUTONOMOUS AGENTS USING GENETIC ALGORITHMS

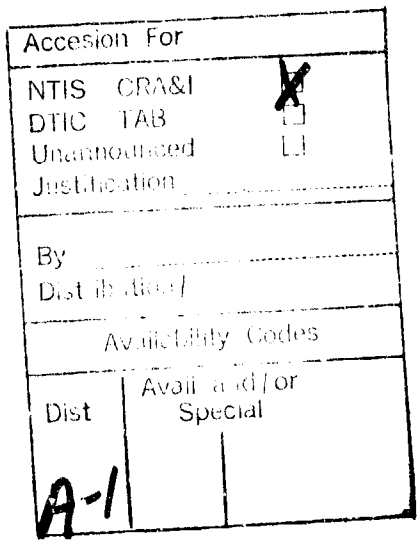
THESIS

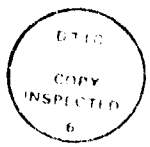
Edward O. Gordon
Capt

AFIT/GE-93D-10

Approved for public release; distribution unlimited

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U. S. Government.





AFIT/GE-93D-10

DISCOVERY LEARNING IN AUTONOMOUS AGENTS USING GENETIC ALGORITHMS

THESIS

Presented to the Faculty of the School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science in Electrical Engineering

Edward O. Gordon, B.S.

Capt

December 1993

Acknowledgements

A document of this nature is rarely the solitary effort of a single individual, and such is very true here. I would like to take a moment to thank all those who have collectively provided the support that made this effort possible.

First I must thank the members of the Genetic Algorithms Research Group for providing a more than worthy platform to air new ideas. Their suggestions guided the directions of much of this work.

I would also like to thank the COOL simulation team for their valuable support and efforts during the designing and implementing of the PDP-C interfacing routines. I only wish their was more time to pursue the efforts. I also offer thanks to Dan of DIS, who provided a stable sounding board for my DIS efforts.

The guidance and suggestions of my Thesis Committee must not go unnoticed, since these shaped the contents of this final document. And the instructors at the Air Force Institute of Technology deserve my best of wishes for some of the best instruction I have ever received. Thank you all.

To my Thesis Advisor, Dr. Lamont, who has done a Herculean job these last many months keeping us advisees on track, no level of thanks is sufficient. Of anyone, his suggestions focused my efforts and made me think of what I was doing and where I wanted to go and his instruction gave me the tools to get there. Thank you for everything and letting this work take me to where it may.

The people of SC Computer Support provided quite excellent computer facilities during my research efforts. Their steadfast dedication to keep it all running and to keep us users happy should not go unnoticed. Thanks.

Finally, of all those in my life who deserve thanks, my parents must top the list. Their support over all these years ultimately made all of this possible and to them I dedicate this effort. My deepest of thanks.

Edward O. Gordon

Table of Contents

List of Figures

Abstract

The control of autonomous agents in changing environments is an ongoing research effort. In particular, controlling autonomous aircraft agents in a simulated environment has taken on new interest as the new Distributed Interactive Simulation (DIS) draft standard evolves into a useful document and distributed simulations are emerging that implement parts of the standard. Efforts at the Air Force Institute of Technology (AFIT) focus on implementing a simulation network that can support interactive air combat and other real-time training needs.

One element of this picture is autonomous aircraft opponents. These simulation agents can provide pilots and others with real-time adversaries that can test their skills in various scenarios. These opponents have typically been human-controlled agents or simplistic rule-based agents that have little ability to adapt to environmental changes. Both approaches have characteristics that limit their use in real simulation networks.

This investigation examines the use of a genetics-based classifier system for agent control. These are robust learning systems that use the adaptive search mechanisms of genetic algorithms to guide the learning system in forming new concepts (decision rules) about its environment. By allowing the rule base to evolve, it adapts agent behavior to environmental changes.

In this investigation we first examine the learning needs of autonomous aircraft agents, showing how multiple learning strategies are possible and that the best approach is a coherent combination of these. We then design a control system using a distributed filtering architecture and a genetics-based classifier system modified to support a phasing-rule niching system based on phase tags. Finally, a prototype system called the Phased Pilot Learning System (PPLS) is implemented based on this design and tested within a limited simulation environment. Results from empirical tests show that this approach is a viable alternative to other control methods.

DISCOVERY LEARNING IN AUTONOMOUS AGENTS USING GENETIC ALGORITHMS

I. Introduction

Autonomous aircraft opponents are opponents in a simulated world that can act autonomously within the context of that world. Such agents can provide a human or non-human pilot with an adversary to interact with. It can be argued, however, that to provide a realistic opponent, the aircraft agent must have the ability to "learn"[1] from its environment, including the other agents that may inhabit it, and use that knowledge to adapt its actions to the context of the current situation. Such adaptation is necessary[2] in long-term scenarios where the ability to update the simulated opponent externally is limited or even non-existent, such as during an extended length interactive session.

If autonomous agents are to demonstrate realistic behavior within the context of complex scenarios, they must respond at many levels, ranging from aircraft interactions (as the agent escaping from an enemy) to executing some form of a mission agenda (such as finding a target and dropping an ordinance). The various learning strategies required to perform this range of actions must be integrated into the system in a coherent and time-efficient manner if the agent is to behave at these different levels of interaction as a realistic opponent aircraft.

This investigation examines the ability of learning systems built on genetics-based learning systems to control autonomous aircraft agents in a simulated environment. We focus on how different learning strategies can be incorporated in such an agent controller to enhance its performance and on the efficiency of such agents in learning to adapt to the needs of the environment. Further,

[1] A working definition of "learning" will be presented in the next section. Until then, the reader should treat the term "learning" as meaning "the modification of a behavioral tendency [of a system] by experience [or other means]," as defined in Webster(83:649).

[2] Except, maybe, in trivial stimulus-response situations where a predefined set of behaviors is always adequate.

we generate an interfacing structure that allows the learning system to connect to and control simulated aircraft agents in differing simulation systems and environments.

This first chapter introduces the above areas of investigation and provides the reader with some initial insight into this complex but intriguing problem. In it we first define the problem being investigated, then detail the assumptions being made, identify the scope of the investigation, and outline the approach and methods being used. We conclude by laying out the content of the remaining chapters.

1.1 Problem Background

This section presents some background on the autonomous agent problem and asome current efforts at implementing autonomous aircraft agents. More detailed discussions of these issues are presented in later chapters.

An area of research currently being pursued both at the Air Force Institute of Technology (AFIT)(29, 39) and elsewhere (15, 39, 27) is that of aircraft simulation. In particular, work continues in building autonomous agents (self-contained simulation components or entities) that can be interconnected in a distributed simulation environment (29, 16, 31, 32). These resultant simulation networks serve numerous purposes. They provide a resource for aircraft pilots to practice their skills in a non-life threatening simulated environment. They also allow pilots to try maneuvers and other operations potentially too dangerous to do otherwise. Such practice and experimentation can increase skill and allow the development of new maneuvers with minimal risk to the pilot. Another use of multi-agent distributed simulation systems is to testbed new aircraft and other systems before actual construction. These simulated prototypes allow new ideas to be tested and refined before the actual bending of metal takes place, potentially cutting costs and increasing the quality of the final product. Yet another use of these simulation systems is in developing and testing new technology to assist pilots of real aircraft. One such example of this is the pilot associate system (39, 29).

The key to such an aircraft simulation is a sufficiently accurate model of the objects being simulated (such as an aircraft) and of the environment in which they exist (such as the space the aircraft will "fly" in). Such models allow the computer software to describe, explain, and predict the behavior of the real world counterparts of the various models(37:5). These models vary greatly in their depiction of these phenomenon in as many ways as there are uses for such models, and typically they involve various trade-offs that depend on factors such as the aspects of the real world object that are most important to represent, the needed accuracy of the representations, and the computer processing capability available. Since humans generally interact in real time with aircraft simulation systems, these models are concerned with exhibiting realistic behavior (within the context of the model) in a time accurate, *real-time* manner. This need for fast (real-time) system response and the resulting communications bottleneck has been one of the major limitations to current distributed aircraft simulation systems (29).

Another limitation of these systems is the lack of adequate opponent modeling (29). In many cases a single user (a pilot) is interacting with the system, typically to practice skills or to train on various tactics. In these single-user applications, an adequate and challenging representation of an opponent is needed in order to test the trainee's various tactics and skills against. These autonomous opponents are software models of real-world potential adversaries (or any other object in the simulated world) and simulate the actions and tactics of such an adversary. Generally these autonomous agents rely on the techniques of artificial intelligence (AI) to "intelligently" control the actions of the opponent model.

Some success has been attained in this area using a hybrid of AI systems However, the knowledge-based reasoning approaches used to give the models this intelligent behavior have generally been based on rule chaining systems that have been too processing intensive for on-line use. This has forced current implementors to develop a compilation of all possible actions and tactics off-line (which has be called a *universal plan* within the AFIT community(28)), and then to encode

this information into fixed tables that provide quick on-line access(70). The major limitation of this approach is that the tactics are, by necessity, fixed and the model can become predictable to the human trainee. Another problem is the inability of such systems to "learn" (adapt to their environment) autonomously. Since the actions and tactics are encoded in a table off line, any improvements to the system generally requires some sort of human intervention. This can become prohibitively expensive (both in time and resources) to do on a regular basis. Another problem with non-learning systems is in their sometimes less than desirable approach to reacting to novel situations not directly addressed by the plan, which can lead to less than desirable consequences (29). Thus, techniques that can provide on-line learning capability to autonomous agent adversaries while allowing such adversaries to function in real time need to be considered in such applications and is a main focus of this investigation.

One can derive a list of desired qualities for a simulated autonomous aircraft opponent.

- The autonomous agent system should realistically portray the particular adversary that is being modeled, including short and long-term behaviors.
- Such a system should act independently with little or no need for guidance from an external control while the agent is interacting with its environment.
- To prevent predictive responses from creating a disadvantage for the agent which an enterprising pilot might exploit, the system should adapt to the environmental demands placed upon it in an intelligent but externally unpredictable fashion[3].
- Such a system must be able to interact with the existing simulations currently in use using standard protocols (such as Distributive Interactive Simulation or DIS)(19).

[3]If the response of an agent is indeed the best it can be (i.e. is *optimal* in some sense of the word), then the agent's behavior should beat all adversaries in all situations the behavior was designed for. However, back a few years ago I saw, on the Brittish TV series Dr. Who, an interesting situation where two robot races were warring, both making optimal decisions and waiting for the "right moment," but neither could make the first move, since both sides continuously predicted the optimal behavior of the other and countered it. This lead to dead-lock. If a stochastic process is not used in such a decision system, then even an optimal system (if it could be built) could become predictable (and exploitable), unless it took into account the ability of the opponent to make a less than optimal move to gain advantage in a complex situation. This was the approach arrived at on the TV series – one side needed to make an intentional error or silly maneuver to confuse the opponent and gain an advantage.

- The agent needs to fully interact with its environment, including other simulation participants, in real time (i.e. with sufficiently fast response to react realistically to the environmental situation).

As noted, these are desired qualities. This investigation is concerned with the feasibility of creating such a system and so all of these desires need not be fully implemented. It is useful, however, to keep a list at hand that shows where one should be heading.

This investigation thus focuses on how these characteristics might be implemented in a learning system based on the genetics-based classifier system architecture described by John Holland(36). Various strategies of learning are introduced to further define the problem and the possible approaches.

1.2 Applying Different Learning Approaches to The Problem

To examine the need for learning in aircraft systems, it is necessary to first examine what is meant by "learning" in general. For the purposes of this discussion we define *learning* as the adaptation of a learning system to its environment in such a way that it can perform its tasks better. This definition carries with it many assumptions (addressed in Chapter 2), but it provides a good working definition for the moment. For this definition to be applicable, the learner must have some defined set of tasks to do (to provide something to measure) and must have some means to interact with some *environment* that the tasks require it to manipulate.

A set of strategies have been defined to describe the different ways one can learn. These are *rote learning* (including *learning by implantation of knowledge*), *learning by instruction*, *learning by deduction*, *learning by analogy*, and *learning by induction*. Rote learning requires no effort by the learner, since knowledge is directly encoded into the learner by some means, bypassing the environment. Learning by instruction is very similar, but uses the environment to provide the learner with the knowledge to learn. Learning by deduction is where the learner applies deductive laws and

approaches to convert knowledge in one form to knowledge in a second form. Converting tables of temperatures from Celsius to Kelvin and reducing logical relations are both forms of deduction. Learning from analogy involves the use of similarities between knowledge representations such that knowledge about one representation (such as an aircraft object within the simulated environment) can be used to predict relationships about another (such as another aircraft). Finally, learning by induction uses generalisation to predict the characteristics of groups of representations. Analogy is a form of induction, since it involves generalizing the characteristics of one representation and applying those generalisations (via deduction) to another. Other forms of induction include *learning by examples* and *learning by observation and discovery*. These are more thoroughly addressed in Chapter 2(6).

We argue that each of these learning strategies are useful in an autonomous aircraft agent. The rote and instruction strategies provide a means to load information quickly into the agent, similar to how a human student uses books and listens to lectures to acquire knowledge in a relatively processed form. Deduction includes calculations of trajectory and transformations on coordinate data. Analogy provides the agent with the ability to reduce the rules it uses to the environment into a smaller and more general form, allowing more knowledge to be squeezed into limited resources. Finally, learning by observation (watching) and discovery (doing) provides the agent with the means to fill in the model that it keeps of the world and provides the basis for adapting to environmental change. But just as the rote methods (rote learning, instruction, and induction) provide the starting knowledge base that reduce the initial learning curve, the inductive methods (analogy, observation, and experimentation) provide the means to go beyond this starting knowledge.

The specific form of induction learning used in the implemented learning system of Chapter 5 is based on *genetic algorithms* (GAs). These algorithms use partial knowledge structures (*building blocks*) to build new knowledge relationships they then can try out on the environment. Based on

a stochastic search, they use *selective pressure* to "weed out" the less useful potential relationships from the better ones. GAs may allow a system to quickly adapt to an unknown environment by trial and error. However, they are algorithms based on random chance and so may not converge at all on a solution. See Appendix A for more on GAs in general. Chapter 4 details how they are used in production-based classifier systems to facilitate learning by new rule creation(22).

1.3 Focusing the Investigation: The Route Finding Problem

This section discusses an application for an autonomous agent interacting with a limited environment. First the target environments are described in brief (a more complete description is given in Chapters 4 and 5), then a specific problem is defined that tests the learning aspects of an autonomous agent and is used to analyse the implemented test system.

The environments. The first target environment for this investigation is the Distributed Interactive Simulation (DIS) environment as currently implemented at AFIT. A developing standard, DIS specifies the communication protocol between the various simulation objects (aircraft, etc.) on a distributed computer network. Network data packets[4] are used by each simulation participant to broadcast its location and other state data to the others on the network. A common terrain mapping is used by all participants, allowing any number of aircraft, tanks, missiles, etc. to coexist in the simulated world. One interfacing goal is to design an interface for our implemented system to the DIS environment.

The second system was developed as part of another research effort. It is a rule-based aircraft simulation in Clips/COOL(31, 32). Entitled "Pilot Decision Phases in Clips/COOL" (PDP-C), this system executes on Sun Sparc workstations. One goal is to interconnect this system with other DIS simulations via network connections, allowing agents within the PDP-C system to interrect with agents on the DIS network. A real-time graphical viewing interface executing on a Silicon Graphics

[4]In the Internet User Datagram Protocol (UDP) standard format used on Unix and other systems.

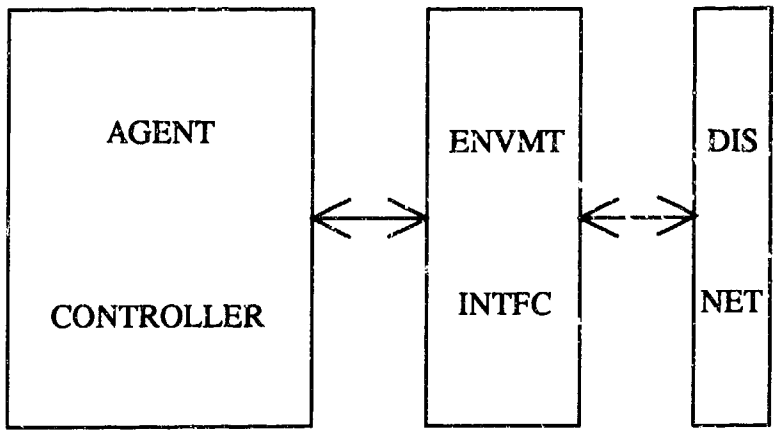


Figure 1.1 Environment 1.

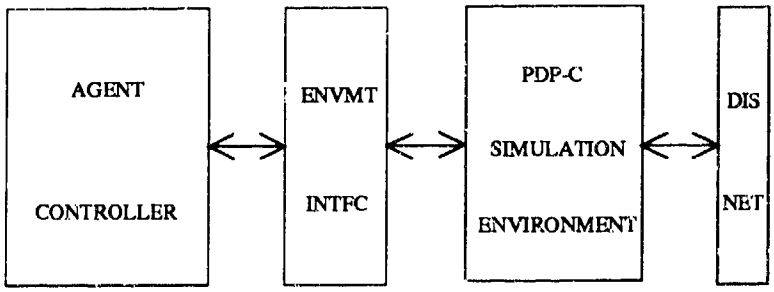


Figure 1.2 Environment 2.

workstation system could then be used to monitor the interactions. The PDP-C simulation system supports four aircraft objects as well as a fifth aircraft specifically designed to be controllable via an external interface. Interfacing to this external communications port is the second interfacing goal for the implemented learning system and should allow for comparisons of the rule-based approach of PDP-C and the stochastic production system approach of the implemented genetics-based classifier system we use as the learning system.

The third interface and environment is an internal test system designed as part of the implemented learning system. This interface and single-craft simulation environment provides a learning testbed to develop the system without the initial overhead of the other environments. The interface design is such that the interface-unique components are isolated to a single package of interface implementation routines. This allows the implemented learning system to train in one environ-

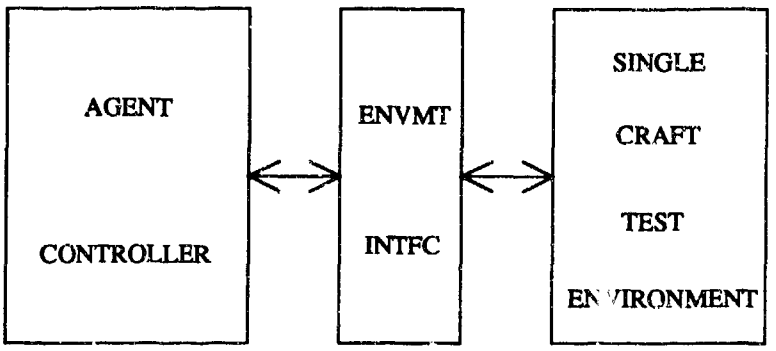


Figure 1.3 Environment 3.

ment and then be moved to another environment to test the training. The goal is to compare the implemented system's performance in each of these environments.

The test problem. One of the objectives of this research is to show that aircraft agents can be constructed that perform a mission autonomously within a changing simulation environment. Due to time and resource constraints, this is limited to a feasibility study and is not required to meet all the goals given in section 1.1, but only to show that such can be met using this approach with minimal change. (See the later sections of this chapter for other limitations on the target prototype system.)

The developed test system is to have the following characteristics.

- The learning system is to demonstrate control of a simulated aircraft in each of the target environments.
- The learning system is to perform a multiple-goal task to show that it is capable of controlling the aircraft and bringing it through an entire mission (sequence of tasks).
- The controlled aircraft is to interact with its environment in a simple but "intelligent" way. Ideally, the agent should exhibit "realistic" (real world) aircraft behavior within the limits of the simulated environment.

- Real-time execution and interaction should be addressed, as the actual implementation of such a controller must interact with other agents in real time.

An autonomous aircraft controller that meets these criteria should be scalable to more complex tasks. This is examined in later chapters.

For simplicity, only one goal is to be active at a time. Though, in a complex simulation environment, multiple goals are expected to be concurrently relevant, this simplifying assumption is needed to efficiently implement the system using the proposed design. Part of the evaluation of this system includes the effects of this limitation.

The basic test problem, which is further detailed in Chapter 5, is defined as follows:

1. Begin at a location in world coordinates declared to be the agent's base.
2. Fly to a target some distance away from the base.
3. Drop an ordnance on the target (assumed to be a ground target).
4. Return to the base (starting point).

To add to the complexity of the task, the system must avoid contact with any hostile aircraft. If contact is made, the agent is to basically " run away" from (evade) the Bogie aircraft. In other words, the mission of destroying the target is the primary goal of the system (i.e. the system is acting solely as a bomber). Further, only minimal controls (direction, speed, etc.) are given to the agent to limit the complexity of the learning task. This is another limitation that is analyzed later.

Time does not permit the addition of fuel and other considerations, though such can be added to the system. Adding other factors, though, increases the complexity of the learning problem and makes the analysis of performance harder.

Analysis of the test problem. The test problem can be considered that of finding a usable route from some location to another through a changing environment. Although related to the routing problem of much fame and which is being studied at AFIT currently using parallel search(17) and

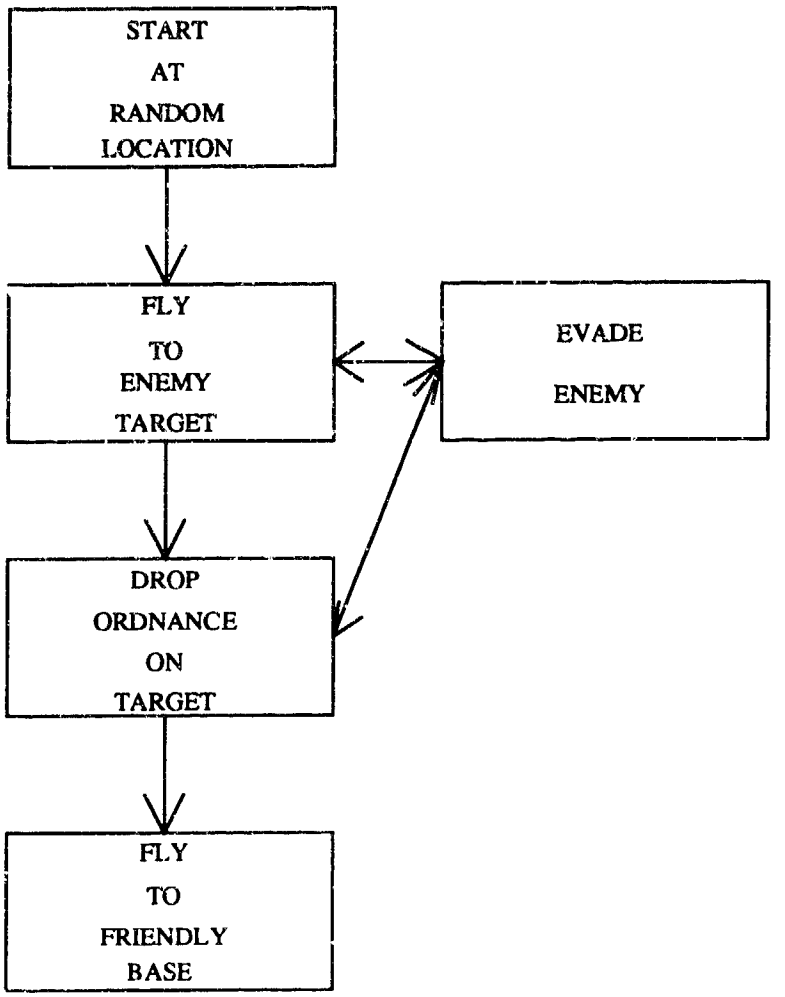


Figure 1.4 The test problem.

parallel genetic algorithm techniques(53), this problem is one of *route discovery*, since the obstacles of the route (i.e. Bogie aircraft) are not known until encountered. This forces the system to adapt the knowledge structures of the system to this changing environment.

Further, this problem has multiple objectives that must be accomplished in a possibly changing order. The system must be able to learn the individual subtasks and later apply this learning to other missions composed of similar (but not identical) subtasks in differing order. For instance, the system might learn to drop an ordnance on one location, then evade an enemy Bogie, on one training mission. Then on a later mission must evade two crafts first before reaching a target at a different location. The subtasks must be learned in a way that allows the reuse of the training when it is appropriate to the mission.

Problem focus. This research effort had the following goals:

- Investigate different ways to apply genetics-based machine learning techniques to the control of real-time opponent simulation systems.
- Determine whether applying such techniques can beneficially improve the adaptive performance of such systems.
- Determine whether such improvements are maintained in a changing environment.
- Determine how these techniques influence the design of interfaces to the target environments.
- Determine how these techniques can be adapted to a mission scenario that requires the execution of a sequence of learned behaviors.
- Determine the scalability and extendability of the implemented system, including the ease with which the system can be ported to a parallel architecture to speed up execution

The approach taken by this research is that learning systems are search processes that show degraded performance as the complexity of the search space is increased. This lead us to defining an interfacing structure that minimized this search space for each learning task presented to the system.

This "limiting of learning" focus is one result of this investigation and is examined throughout this thesis.

1.4 Scope

This investigation concentrates on three distinct areas concerning autonomous aircraft agents: (1) the learning requirements of such systems, (2) how genetics-based classifiers compare to various alternatives available for implementing such systems, and (3) a design that represents one potential approach to meeting the needs of an autonomous aircraft agent. The need for adaptation in autonomous aircraft agents is emphasised, and is demonstrated by building a prototype system based on a genetics-based discovery learning system. The prototype system provides a way to illustrate and evaluate the concepts proposed.

The two major types of classifier systems[5] were analysed and compared. These systems use different approaches, however, and comparisons were limited to the applicability of each system to the target task. No other genetics-based learning techniques were considered, except as noted in the literature review. Neural networks especially are concept learning systems that show promise in this area, but were explicitly excluded from this study since they are not rule-based systems. Hybrid systems (a mix of the two classifier methods for instance) were analysed, but were not needed to build the basic system. The availability of parallel implementations of selected systems was also considered, and played a key factor in determining the scalability of the overall system to more complex and computationally intensive autonomous agent systems.

Finally, this is a feasibility study. Since the prototype system is a demonstration system, the created agent is not intended to implement a full set of aircraft agent behaviors. Instead, the goal is to design a system that could be scaled to more complex systems based on distributed extention of the design.

[5] "Michigan" and "Pitt" – see Chapter 3.

1.5 Standards

This research used the DIS draft standard, current version (2.0), as a guide to designing the interface to the simulation network here at AFIT(16). Other applicable standards include various software standards relevant to this effort (as presented in the courses noted below) as well as other AFIT guidelines as appropriate. All efforts are to be documented in accordance with established research procedures.

1.6 Approach/Methodology

Preliminary. Research through most of the 1993 calendar year was facilitated in related courses, namely the CSCE 686, CSCE 656, and CSCE790 algorithms courses, the CSCE 692 and CSCE792 computer architecture courses, and the CSCE523, CSCE623, and CSCE624 Artificial Intelligence courses. Prototype systems were built up in these courses and the results used to test the feasibility of these techniques in addressing the autonomous agent problem. The final system was determined from this experience as well as the literature search. Availability of software was of significant concern in this choice. The software design methodologies of CSCE593 and the principles of Object-Oriented Design (as identified in applicable references(7)) were used in the design as much as possible. This was limited, though, by the current design of existing software used for this effort.

The prototype system uses a modified "Michigan" system design based on the CFSC-1 public domain classifier system written in the "C" language by Rick Riolo of the University of Michigan(59). The simulation environments were described previously. A test utility was developed to provide a view of simulation activity using a simplistic vt100-compatible screen on a Sun workstation. Other plotting facilities, such as a GnuPlot interface, are used to generate output tracks.

Real-time interfacing to DIS is being pursued independently and so only a proposed design is addressed[6]. The design presented, however, allows for later connection to the proposed DIS interface design via an interfacing layer which can be modified to whatever to map the PPLS system interface to whatever form the DIS interface takes.

System Overview. The final system developed, called the Phased Pilot Learning System (PPLS), uses various techniques to develop subpopulations that address the different behavioral needs of the agent. These techniques include various *niching* strategies that promote multi-modal populations(13) (populations that converge on multiple rule clusters) to develop. These subpopulations of rules *species* allow a single population to adapt to different types of environmental situations, with each species of rules addressing a specific need of the agent. A phasing system is implemented that adds an additional higher level of population isolation allowing each population to develop independently the species needed to execute each part of a mission. Although the phasing mechanism isolates the populations, the rules of other populations are available to seed the learning in new task domains. Chapter 4 develops this approach more fully.

The interfacing between the learning system and the environments use a layered approach to isolate implementation and promote portability between the environments. All software, except for parts of the PDP-C interface (which is in Clips COOL (Clips Object-Oriented Language)), are in the "C" language and are generally portable to other systems (such as a PC environment, if the networking is available). This includes a "C" interface between the COOL object system and the CFS-C routines. CFS-C itself is written in "C" using a functional breakdown and is designed in a modular and portable manner, which facilitates system design and enhancement.

[6] This research is just beginning and is slated for completion in October 94. No documentation of this effort (other than verbal communication with the researcher) is currently available. However, much of the design specifics of the interface were worked out to be compatible with the proposed interface design.

1.7 Materials and Equipment.

Access to simulation source code and work station facilities were required. Arrangements were made to use the AFIT/SC Silicon Graphics facilities in this effort. These are multiple-processor systems well suited to this task. The needed software for DIS support is currently designed for use on either a Sun workstation or a Silicon Graphics system; however, most of the DIS network activity is on the Silicon Graphics network, so porting of the implemented system to that network may eventually be required to network to other ongoing DIS projects. Such porting would allow the PPLS agent to interact with other agents in a relatively complex environment. This porting was not done for this research effort.

This research was coordinated with other research on DIS systems being done at AFIT. This includes the PDP-C research efforts, as well as those working to interface these systems to the DIS network. I emphasize this since any efforts should benefit from mutual exchange of ideas, as well as the research opportunities provided by building compatible and extendible interfaces between these various efforts. Any serious effort to address the implementation of a realistic DIS simulation network would benefit from such cooperation among those researching different aspects of this complex network system.

1.8 Summary and Thesis Layout

This chapter has presented an overview of what is being investigated. Learning strategies were introduced that can be applied to problems in machine learning and we have showed that all these methods have some bearing on the autonomous aircraft agent problem. Limiting a particular autonomous agent system to only one of these strategies reduces greatly the system's potential to react appropriately within the constructs of its imposed environment. For instance, implanting rote knowledge in an otherwise inductive-based system allows the system to avoid a stretch of learning-induced mistakes that might make the agent impractical in normal usage. We also argue that

a hybrid of these methods must be brought to bare if effective and efficient autonomous aircraft agents are to be realized.

The autonomous agent problem can be observed from different views, depending on the requirements of the target system desired. For realistic behavior in a complex and changing environment, it is proposed that a learning system must have the ability to learn by experimentation, observation, and discovery if the autonomous agent is to adapt to the activities of the other agents in the environment in a way that allows the agent to progress toward its goals without external intervention. Otherwise, other agents, some of which may be human pilots, may find weaknesses in the agent's behavior based on the predictability of its actions and exploit them[7].

The other main advantage of providing the aircraft agent with the ability to learn by discovery is the potential for the agent to discover and improve on techniques that allow it to better operate in its environment. Such activity removes much of the burden from a software engineer, since the learning system can, in many cases, teach itself. And such discovery learning could reveal new techniques or perhaps weaknesses in old techniques.

The rest of this thesis is organized as follows:

- Chapter 2 provides a review of the theoretical literature relevant to this effort. It introduces many of the concepts we build on in later chapters and demonstrates that providing a learning system with the ability to use many of the different learning strategies gives the learning system a better ability to efficiently and effectively interact with its environment.
- Chapter 3 continues the literature review, but focuses on current implementations of autonomous learning systems and agent controllers, their strengths and weaknesses, and their applicability to the autonomous agent problem. The chapter focuses also on autonomous

[7] I must emphasize that even discovery learning systems make mistakes – indeed, it's the primary way such systems learn. The point here is to learn from the mistake and not repeat it (too many times). A system that can never learn from its mistakes is at a definite disadvantage in an adversarial environment.

agent behavior and genetics-based machine learning, though other methods are briefly mentioned.

- Chapter 4 focuses in on the design of a proposed learning system from a theoretical point of view. The chapter examines the different types of inductive learning that can be applied to (standard or "Michigan" type) classifier systems. Cover operators and the genetic algorithm as used in rule discovery are presented, and various ways to limit the "detector" and "effector" domains are discussed. A new approach to "niche" formation based on dividing the rule population into partially isolated *phases* is presented and serves as the primary mission task level control mechanism in the PPLS system detailed in Chapter 5.

- Chapter 5 presents the details of the Phased Pilot Learning System (PPLS), which is a modified version of the CFSC-1[8] classifier system. This chapter outlines how the program was analysis and design and also how the learning system is interfaced to the environments it can interact with. We emphasize the use of context limiting and distributed subagents to implement the system. By making each component of the system simple and loosely coupled, the system as a whole can be easier to maintain and to scale to more complex systems. It also is easier to parallelize.

- Chapter 6 details the internal test environment and related tests using it. We show that the system is capable of progressing through the test environment using implanted rule sets. Then the effects of the discovery operators on effective rule sets and on partial rule sets (where some critical rules have been omitted) are examined. Tests are also performed with completely random rule sets to judge the effectiveness of the reward system and the discovery operators in evolving effective rules. A discussion of implementation issues and an analysis of the environment interfaces is also provided.

[8] Classifier System in "C" - a classifier system package discussed in Chapter 2 and addressed from a design perspective in Chapter 4.

- Chapter 7 reviews the major points of this investigation, including the realism of the test model, the learning system's ability to control the aircraft agent and to autonomously learn, limitations on this learning and what can be done to improve it, and the effectiveness of the interfaces. The practicality of the system and its scalability both to larger learning tasks and to more complex environments are also examined. The chapter ends by summarising the conclusions of this investigation and suggesting areas for further research that might prove fruitful.

II. Literature Review

In this chapter we summarize and analyse applicable current theoretical literature. First, we more fully define what we mean by "learning" and present a working definition used in the remainder of this thesis. Then we provide extensive definitions for the learning strategies presented in Chapter 1 and apply them to the autonomous agent problem. The analysis shows that all forms of learning are useful to the development of autonomous aircraft agents. We discuss this issue and note what happens when one or more forms of learning are left out of a system.

We also review the autonomous aircraft agent task and derive evaluation criteria useful to the problem. The criteria developed are used to guide the design of the prototype system developed in Chapter 5. We follow this by presenting some of the basics of concept learning, an area that is intertwined with inductive reasoning and which is a central idea that flows through this entire thesis.

Finally, we analyse the autonomous aircraft agent problem as a task of computational search, defining the domain space, the solution space and the nature of the operators. We extend this notion to one of an adaptive search within these spaces. We note that the task of controlling an autonomous agent can be mapped to the task of searching for valid concepts (decision rules) describing appropriate actions within the simulated world and then applying them.

2.1 Defining Learning

To examine the need for learning in aircraft systems, it is necessary to first examine what is meant by "learning" in general. This section presents some definitions of learning and chooses one that facilitates measuring the learning taking place in an autonomous aircraft agent. Then we present a taxonomy of learning in which to view the aspects and needs of the agent.

2.1.1 The many faces of learning. To better understand what is meant by learning, we begin here by looking at some definitions of learning[1]. The idea of *concept learning* and how it applies to the autonomous agent problem will be formally addressed later in this chapter.

As Carbonell, *et al.*, explain(6:3)

> Learning is a many-faceted phenomenon. Learning processes include the acquisition of new declarative knowledge, the development of motor and cognitive skills through instruction or practice, the organization of new knowledge into general, effective representations, and the discovery of new facts and theories through observation and experimentation.

It is no surprise, then, that the types of learning processes that prove most effective for a particular problem depends markedly on the objectives of the problem. For instance, in applied learning systems, such as many robotics systems, the objective is to perfect the skill of a system in accomplishing a simple or complex, but predictable, task. For other systems, the objective may be to acquire knowledge about a task domain either represented as a set of facts to process or, perhaps, an environmental model to explore. This diversity of task and objective, however, makes the precise defining of what we mean by learning a quite nebulous proposition. Yet to measure the success of a learning approach such a definition is necessary.

One measure of success used by many is a simple measuring of how well a system improves as time goes by However, saying simply that, as a system improves, it learns, can be misleading. For instance, Michalski (49:10) notes that, ". . . wine improves with time, but nobody would call such an improvement learning." This has focused the definition debate on finding a more specific measure of learning, such as one based on *improvement criterion*. Simon(74:25) has come to the following definition of the learning task:

> *Learning denotes changes in the system that are adaptive in the sense that they enable the system to do the same task or tasks drawn from the same population more efficiently and more effectively the next time.*

[1] The following discussion is meant to be general enough to apply to any learner, not just to machine learning systems.

This, in essence, defines learning as simply the tuning of a system to a specific task. It also implies that the efficiency and the effectiveness of the task can indeed be measured, which is not always clearly the case[2]. The perspective of the observer might not attribute the same significance to the various features of the environment that the learner does, for instance. Or the learner might be interacting in a social situation that guides behavior in ways not obvious to the observer.

Another definition presented by Michalski quotes Minsky (1985)(49:10) as defining (human) learning more generally:

Learning is making useful changes in our minds.

but Michalski notes that even Minsky thought this definition too general for practical use. Even so, this reflects the view of many that learning is an internal (human?) process and, as such, is intrinsically hard to quantify.

Michalski himself settles on the following definition:(49:10)

Learning is constructing or modifying representations of what is being experienced.

In this definition, *experience* is defined as any sensory stimuli, as well as any *Gedanken* (internal) processes that provide input to the system. Note that the emphasis is on the building of an internal representation rather than on performance. Michalski further defines (49:11) three criteria in evaluating such constructs: ***validity***, or the degree of accuracy that such a representation fits reality; ***effectiveness***, or the usefulness of the representation in achieving the goals of the system (and, as such, is an indirect measure of performance[3]); and ***abstraction level***, or the scope of detail and the explanitory power of the representation.

2.1.2 A taxonomy of learning. The act of learning has been classified into six strategies(49): *rote learning*, *learning by instruction*, *learning by deduction*, *learning by analogy*, and *learning by*

[2] An example of this might be trying to determine how the sometimes quite inefficient actions of one ant of an ant colony might, in fact, be the most efficient action it can do in that situation to maintain the colony as a whole.

[3] A formal definition of *efficiency*, one measure of performance, is given later in this chapter in the section on concept learning.

induction (which can be decomposed into *learning from examples* and *learning by observation and discovery*). This taxonomy provides a useful framework in which to view the autonomous agent problem and is worth examining here. The following is mainly from Carbonell, Michalski, and Mitchell(6).

Rote learning and direct implanting of new knowledge. This strategy can be considered nothing more than direct programming, since knowledge is either "hard-wired" into the system or provided as a simple database or look-up table. Standard computer programming (coding) is a form of direct implantation. Memorisation is a form of rote learning. Note that no transformation of knowledge is performed in this strategy – the knowledge has to be provided in a form directly usable by the system.

Learning from instruction. Learning from instruction (or learning from being told) differs from rote learning in that the learner receives the new information via some external source (such as a teacher or textbook) and transforms it into an internal form that can be integrated with previous knowledge. The key difference is that the learner here performs its own receiving, storage, and integration while it continues its interaction with its environment.

Learning by deduction. Here the learner draws deductive relations between the data in the system. This includes knowledge reformulation, knowledge compilation, macro-operator creation, chunking, and other transformations using a data reformulation. Examples include the conversion of a temperature to another temperature scale as well as the chunking of rules as in the Soar architecture (43).

Learning by analogy. Inductive learning involves the generalisation of knowledge into useful concepts that can be applied to other situations outside the immediate context that the input data was acquired. Learning by analogy involves two steps. First a generalisation is made about knowledge already existing in the system (induction). Then this generalized knowledge is applied to other, not directly related situations (deduction). This form of learning is imprecise, since the

generalisation may not be completely accurate. Note that *learning by being reminded*, where stored knowledge is applied to a situation based on the recognition of some perceived trigger, is a form of analogy.

Learning from examples. This is a special case of inductive learning where the learner must induce general concepts given a set of examples possibly including counterexamples. Since the examples may have no direct relation to each other, this form of learning is more general than learning by analogy. Also, the examples may come from a teacher, from the learner itself (as when it forms and tries various possible relations between previously acquired data), or from the external environment (which are essentially random examples since the examples are not controlled by the learner). Learning by examples is usually an incremental process, with the learner's internal data structures being updated after each example, but could be batched process also (all examples presented at once[4]).

Note that negative (countering) examples are needed if overgeneralisation is to be minimized. Otherwise, the choice of examples, and the generalisation process, must be carefully constrained to permit only the minimum amount of generalization.

Learning from observation and discovery. Also called *descriptive generalization*, this learning strategy focuses on regularities and generalities that "explain" observations made about the environment. Included in this form of learning are *conceptual clustering* (the forming of object classes describable by simple concepts), classification construction, equation fitting, and behavior prediction of objects in an environment. Note that *genetic algorithms* and *empirical prediction algorithms* can be viewed as falling under this learning strategy.

The level of interaction of discovery learning strategies with the environment can be *passive*, where the learner classifies observations based solely on what is seen in the environment, or *active*,

[4]Some learning systems require a set of examples be available from the start. For example, neural networks are generally trained by repeatedly presenting a set of examples to the learning network until an acceptably low error rate is achieved(57:498). Any system that requires repeated exposure to a set of preclassified examples can be considered a *batch* system in this context.

where the learner interacts with the environment and observes the effects of the actions performed. In the active case, a form of payoff or other guiding mechanism can be used to provide feedback to the learner or, alternatively, the learner can determine the desirability of the effects internally.

Of these strategies, learning by observation and discovery typically requires the most induction on the part of the learner, since little or no guidance is being supplied by the environment itself. Yet this strategy is also the most autonomous, since, by the same token, the learner demands the least from the environment. It is this strategy, therefore, that perhaps shows the most promise in implementing an autonomous agent that can adapt to its environment.

One can further classify learning by the types of knowledge needed and the structures maintained(6). These include ***parameters in algebraic expressions***, ***decision trees***, ***formal grammars***, ***production rules***, ***formal logic-based expressions***, ***graphs and networks***, ***frames and schemas***, and ***computer programs and other procedural encodings***. Of these, this investigation focuses on production rules of the condition-action type, though knowledge represented in one form can generally be converted to any of the others if the representations used are general enough. The use of hybrids of these and other structures are also possible, though they are not considered.

2.2 Controlling Aircraft Agents

This section looks at how the learning strategies described in the previous section provide insight into the implementation of an autonomous agent controller. Of interest to this investigation is the learning needs of such a system and how to best meet them.

Currently many methods are being used to control aircraft agents in simulation environments. Since the control of an autonomous agent of any kind must use some form of "intelligence" to select and guide the actions of such agents, these methods can be mapped to the learning strategies discussed in the last section. This section takes a look at some of these mappings[5].

[5] As with the previous section, see Carbonell, *et al.*(6) and Michalski(49) for a good introduction to the different strategies of learning referenced below.

This thesis effort focuses on a rule-based approach to knowledge encoding. The reader should note, however, that much of what follows is independent of the representation of the knowledge itself.

Rote learning and direct control. The simplest form of control for an autonomous agent is direct control via a set of control rules. These rules, predetermined and implanted into the agent's control structure, allows the agent to react in a predictable way to the current state of the system. Thus, if the control structure is taken to be a set of rules, where each rule matches a unique predefined state (based on the state of the environment and, possibly, some internal state representation, i.e. memory) and prescribes a predefined action when such a state is detected, then such a system functions within the confines of a finite automata. Given a sequence of environmental states, then, the state of the system is completely determined.

The control structure can be represented as an explicit set of rules, but it can also be delineated as a control program in some programming language, as a network of graph nodes that each represent a state the system can attain, or even as a collection of circuits that interface to electronic detectors and mechanical effectors (as in a small robot). What is important is that this representation effectively map all possible states that can be discerned into an action space that is effective in controlling the application[6]. It is this mapping, from a *detected state space* into an application's *action space*, that defines the agent's behavior in the environment it finds itself.

Updates to the mapping that govern and define an autonomous agent with this structure must be accomplished via some external programming mechanism. In the case of a set of rules, the updated rules must be loaded by some outside agent that has the capability to either acquire or create such rules. In the case of a program, some external agent must modify or replace the code to be changed and, if necessary, restart the application. This process may or may not involve suspending the actions of the autonomous agent and may or may not force a modification of the

[6]For this discussion, an *application* can be considered the instantiation of an agent within the confines of a particular environment that it must function in.

current state of the system. In the typical case, however, where the instructions to the agent are essentially "hard-wired" into the system, the changing of the agent's program must be done off-line, forcing the agent's interactions with the environment to be discontinuous and possibly its behavior to be disjoint (as viewed from the environment).

Learning by instruction. In learning by instruction, the agent's behavioral mapping is modified by supplying the agent with information in a form that it can assimilate (convert to an internal usable form that is integrated with other knowledge previously encoded into the agent). The key difference between this strategy and the direct learning described previously is the addition of an interface that the agent can use to add knowledge to its knowledge structure while the agent is still interacting with its environment (i.e. while the application is still running). The information provided by the supplying agent still must be accurate and specific for the agent to assimilate it, and no changes other than those dictated by the transferred knowledge are made. Facts and procedures are memorized, to use a human analogy, and applied as directed in the teachings with no change. Though selection (refusal of unneeded or unwanted information, for instance) and reformulation (conversion of information into a usable form) can occur in these systems, no new knowledge is formulated by the activities of the learner alone.

In the case of aircraft agents, instructed learning is preferred over direct implantation if the system must continue to interact with its environment on a mostly continuous basis. This may be especially important in a simulated environment where long-term activities progressing over a relatively long span of time, and directly involving the agent, are best not interrupted whenever a small change to the autonomous agent in question is needed.

Deduction in the aircraft environment. Deduction can be considered the repackaging of knowledge using logical or other deterministic transformations(49:14). Included here are coordinate system conversions, predicted flight time processing, and other tasks that require the processing of information by strictly "cookbook" methods. This form of learning produces equivalent or more

specific formations of knowledge, such as when detected coordinates given in X-Y-Z form are converted to spherical coordinates (say, to better calculate an intercept angle) or a specific missile flight time is deduced via a set of rules (formulas) that calculate that quantity(6).

This strategy also includes the use of logical transformations, macro-operation creation, and information chunking, which can assist an agent in handling a complex environment, as well as the categorising (sorting) of data, as might be done to maintain a data base of known simulation participants. It does not, however, include the applying of such knowledge out of the context in which it was created. As such, deduction may be of limited use (besides the transformation functions above) in aircraft agents when used alone (not combined with some form of induction, described below)(6).

Analogy. Analogy, or the exploiting of similarities between knowledge structures, is an important mechanism in small systems that must deal with complex environments. In a rule-based system, for instance, analogy can sustantialy reduce the number of rules needed to model an environment effectively by finding common substructures (features) between, say, two types of aircraft and using those common features to create a small number of general rules that apply to both these aircraft. By combining deductive and inductive inference, the system finds substructures within its data and attempts to map the substructure onto a different structure to predict properties of the new structure.

Machine learning systems that apply chunks (subroutines) derived previously to new situations are performing a form of analogy in that the uses of these chunks is being generalized outside the context in which they were created. These "building blocks" form useful groupings of knowledge that can thus be used to improve the system's interaction with its environment. Such systems are therefore capable of generalizing their model of the environment to some extent.

A major limitation of analogy, however, is its reliance on previous cases and knowledge to draw upon. Without such experience or knowledge, there is nothing to draw similarities from,

which leaves the system to try random actions and attempt to build the experience necessary. In an environment where most objects are unique, this form of learning may be limited.

Inductive learning. The most general learning strategy for an autonomous aircraft agent is inductive learning. This strategy includes learning from examples (either supplied or generated) and by observation and discovery. Passive learning (described in the previous section) provides a way for an agent to study its environment and build a model of what it perceives. In the case of an aircraft agent, however, a more active approach results from the agent's interaction with its environment. Inductive learning allows the agent to categorize and learn from the various causes and effects present in its environment. For the sake of this research, the active forms of inductive learning are grouped under the term ***discovery learning.***

Genetic algorithms, search algorithms based on adaptation and natural selection, can be used to promote discovery learning in simplified rule-based systems called ***classifier systems***(36). Many different architectures have been developed showing the viability of classifier systems in various learning domains. As such they represent a good starting point in building a better autonomous agent control system[7].

2.3 Reviewing the Autonomous Aircraft Agent Problem

This section reviews the autonomous aircraft agent problem from the perspective of a problem in learning. We first characterize the environment of the problem. Then a set of criteria are derived to provide a measure to apply to the approaches reviewed in later sections.

2.3.1 Levels of Learning in an Autonomous Aircraft Agent. The autonomous aircraft agent problem can be broken down into two levels of concern: a low-level reactionary problem where the aircraft's main concern is survival in a hostile environment and a higher-level mission

[7]See the later sections of this chapter for more on the applications of classifiers to learning problems. In addition, an introduction to genetic algorithms and classifier systems can be found in Appendix A.

execution problem where a set of mission tasks are to be accomplished within some time constraint[8] For simplicity, we assume that these mission tasks are to be accomplished in some defined sequencial order[9].

The mission portion of this problem, then, is to step through a sequence of tasks. This set of tasks, however, will most likely have prerequisite conditions that must be met (e.g. the craft must be over a target before it can drop an ordnance). This requires the agent to check to make sure that the conditions of an action are met before attempting the action. It may also mean verifying that an action had the effect intended (that the ordnance fell on the target, to continue the example). Thus the actual sequence in which a set of mission tasks are attempted depends on the context of the environment in which the aircraft finds itself.

The moment-to-moment interaction portion of this problem, in contrast, includes those things that basically range from annoyances (a moved target, for instance) to potentially catastrophic surprises (such as the arrival of four enemy aircraft on the scene). Such occurences must be dealt with for the mission of the aircraft to succeed. In the case of a moved target, the intrepid agent must modify its course to move its projected position to the new location; in the case of a surprise encounter with enemy Bogies, a major change in tactics may be necessary. In a way, these actions are the same as the mission tasks of above, but with the difference that these are unplanned actions and perhaps not directly specified by the mission tasks of the system.

In any event, the successful aircraft agent must handle such situations if it is to successfully complete its mission in a complex and changing environment. And, though a set of simple stimulus-

[8] Real fighter aircraft missions are arguably much more complex than this simple model. For instance, as targets are discovered or are destroyed by other means, the missions of the craft may change. However, I believe this model sufficient to portray somewhat realistic behavior within the test environment described later.

[9] Again, this is a rather imposing simplification of the problem, since the actual order of task execution could vary as the opportunities that the craft has to execute them varies with conditions in the environment. Nonetheless, a real preplanned mission of the sort assumed here would not vary markedly from that represented here. For instance, the craft might launch, travel through a set of way points, perform some mission, then return to its base and land. Additionally, alternative mission choices could be encoded into the system, as in real missions, in case a primary mission of the agent became unattainable in some way. As an example, most bombing missions, to my knowledge, specify multiple bomb drop sites prioritized in some way and allow the crew to select the site based on what is happening.

response rules that is keyed to the various situations the system finds itself in may be sufficient to control the craft in all situations (given sufficient definition of the conditions of each state), the system must still distinguish which rules are appropriate to the given situation (decide, for instance, whether to fly to the target or away from the enemy aircraft). And this distinguishing capability must not be forgotten, even if exceptional situations (such as making contact with an enemy aircraft) only happen on rare occasion.

2.3.2 Other concerns of autonomous agents. Brooks notes that the concerns of an autonomous agent are quite different than those of chess playing programs. For instance, the various functions of each part of a system may each have different criteria in what and how they should learn. Trying to learn all these functions with one algorithm may be asking a bit much. Also, the traditional domains of artificial intelligence research isolate the learning system by imposing either a structured or a very limited environment. Autonomous agents must be able to adapt to more complex domains, even though the mechanisms used may be simplistic in themselves. A distributed approach seems the best attack to the problem.

And this is not without examples in nature to back it up. Animal systems are greatly distributed systems that have various sensors and preprocessors that provide the core reasoning mechanism with highly filtered data. A squirrel, for instance, does not have to learn how to visually detect an acorn or discover (at that moment) how to maneuver its arms. It "reasons"[10] in terms of the objects of nut and branch and how moving to a specific location (in a rough way) allows the gathering of the tree's fruit. Applying this to the autonomous agent problem, it can be seen that very simple behaviors are sufficient if each agent in the system (sensors, activators, etc.) provide the reasoning mechanism with just the right inputs and controls to make the problem "simple".

Another issue is that of planning, a "higher-level" function that allows an agent to form sequences of behaviors that allow it to achieve a goal. Agre and Chapman(1) argue that plans are

[10] If in a "primative" way.

not necessary for sensible action if the environment provides the cues for the next action. They also note that plans are best treated as recipes that represent the activities associated with objects in the environment, not necessarily a "program" to be executed. As such, they don't always directly translate into a set of actions for the agent but are just guidelines that offer suggestions to try.

They also note that planning in a changing environment is a continuous interpretive effort and is hard (computationally wise) to do to any depth. This is not to say that planning is not useful to the agent. If a ***static assumption*** can be made about current environment[11], then sufficient time (computationally) may be available to look ahead in a useful way. This is very much dependent, however, on the concepts that remain valid about the environment over the period being planned for. Plans are assumptions about the agent's world state and how to best deal with it to reach some goal.

How to handle a changing environment can be done in different ways(1:29). For instance, ***interleaved planners*** provide the agent with a plan it tries. When it runs into trouble, the planner makes a new plan for the agent. The problem here is recognizing when the plan is in trouble. Waiting until something goes ***really*** wrong (the agent is shot down, for instance) may be a little too late. Related to this is the idea of ***improvisation***, where the agent tries to make the best of the situation. An alternative is to keep going back for plans, which assume the planner knows what it's doing (has a sufficiently accurate model of the world). In a rough way, discovery-based learning systems tend to the former approach, while preprogrammed systems to the latter.

Reactive planning uses a different approach where the agent presents an environmental state to the planner, who then chooses a plan to fit the situation(1). A bit of dispute has surfaced as to whether this is planning or just reacting. If the sequences of actions proposed are taken as suggestions, then they form a plan. However, if they prescribe a set of actions that must be

[11] Which I define as assuming that the objects of relevance in the current environment can be taken as sufficiently static to allow the use of planning. For instance, when we drive to work we assume the roads are as they were the previous day. A planned route is therefore quite useful. Now if the road crews emerge on the scene, this assumption and so the plan may no longer be valid.

carried out, they become more of a macro facility. Even here, however, they represent a plan in the sense that they predict the attaining of some goal if carried out. And they still depend on the interpretation of the environment provided by the agent.

Planning may be necessary to perform complex actions in the agent's environment. The question is how much, and of what kind, is actually needed. A distributed agent may be able to perform quite well on a reactionary basis, if the learning system driving it is given, by the subagents that process the inputs and outputs of the agent, a simple enough model to deal with.

Indeed, the design in Chapter 5 proposes the use of a highly distributed and loosely coupled model to implement the learning system and its interface to the outside world. This is nothing less than applying the well-known concepts of object-oriented parallel programming to simulation(45:179). We emphasise here, however, the need to make the task of each agent (component) of the system as simple as possible so that the complexity of the entire system can meet real-time constraints. This is especially important for the learning system, since learning simple concepts based on a small input domain space and a small action space is much easier than learning concepts in much more complex search spaces. (Concept learning is addressed shortly in Section 2.4.2.) In short, we argue that simple is better, even at the cost of many rather than few agents (components, interfaces, etc.)[12].

2.3.3 Formalizing the autonomous aircraft problem. We now reiterate the problem described above. For this investigation, the autonomous aircraft agent must perform the following:

- Be able to perform a set of tasks defined for the system by some external agent. This set of tasks will be referred to as the agent's *mission*. Instructive learning may be used here, where the information is passed to the agent in some form from another agent in the system. Or

[12]This is from a logical perspective. How the system is actually implemented on hardware is not addressed. However, if the interfaces are simple enough and coupling is minimised, then efficient implementation of the agents on different physical processors is possible and can make a complex application feasible. The communications costs and routings would have to be examined carefully if distributed computer hardware is used, however. We address this topic again in Chapter 7.

rote learning may be used, where the rules that determine the goals of the agent are preloaded into the system.

- Autonomously learn behaviors necessary to navigate the given environment and improve performance on mission operations over time. A *mission operation* is defined as a particular, coherent task set that forms part of the mission, such as flying to a fixed point or evading an adversary. The goal is for the agent to learn efficiently while avoiding catastrophe (i.e. death). Alternatively, the system could be loaded with a preliminary set of rules, and so the task then is to maintain proficiency and possibly improve upon the rules.

- Be able to switch between mission operations without "forgetting" how to do a previously learned mission operation. This capability is necessary to allow the agent to learn from experience how to perform various mission behaviors and use this learning at a later time. Once a sufficient variety of behaviors are learned, the agent should be able to perform variations of these missions in a "skilled" (i.e. "intelligent") manner with minimal error.

Each of these represents a significant agent ability that would be needed in a real-world example.

In addition, the environment for this investigation needs to have the following qualities:

- Be sufficiently complex to model the real world within the limits of the agent's perception. The required behavior should include a *reversal* problem to test the ability to retain knowledge in contradictory settings[13].

- Interface (as a goal) to the DIS network, either through an intermediary, such as the PDP-C simulation environment, or through a direct DIS link.

[13] A classic problem used to test classifier systems, the *reversal* problem trains the learning system on a task using a set of positive and negative rewards. Then the rewards are switched. The rules in the system are now completely wrong and the system essentially has to relearn the task from scratch. After multiple switches, however, a learning system with some form of context-sensitive memory will recognise the switch quickly and load a set of rules from memory for the current task, thus reducing the errors made and the time needed to relearn the task after each switch (84). It is this type of ability we refer to.

2.4 Some Definitions

In this section we more precisely define learning in autonomous agents. Then we define what is meant by a ***concept***, a key idea in inductive models.

2.4.1 The Embedded Agent Model. An autonomous aircraft agent can be viewed as an agent embedded in some environment (the simulation in this case). This *embedded agent* can be decomposed into three components: a *transducer* that provides the agent with environmental information and also provides the means for the agent to effect its environment, a *learning module*, and a *planner*(14:2 – 3). Real-time interaction between the agent and the environment is assumed and other agents, such as teachers, are considered just a part of the environment.

Using this framework, we view an agent as a learning-planning system that receives incomplete information about its environment via its detectors and only has a limited ability to effect that environment via its effectors(14:3). The learner part of the agent classifies the inputs it receives and generates the agent's actions. The planner (if present) can determine and prioritise what experiments to accomplish next and generally influence the agent's exploration and reaction to its environment

A *rational* agent is one that chooses actions that maximise the agent's expected utility, while a *limited rational* agent must divide its time between this activity and actually executing actions. An *autonomous agent* operates independently of human intervention or, more specifically, "... it does not require inputs (except for its initial state) to tell it what its goals are, how to behave, or what to learn(14:4)." In this context, the learner provides the planner with the information in whatever form that it needs to maximise the system's expected utility per unit time[14]. The task of *autonomous learning*, then, is to learn a close enough model of the world in as short a time as possible to allow the agent to maximise its performance(14:4).

[14]This definition is similar to that for *on-line performance* used elsewhere in this document and can be considered equivalent for this discussion

What to learn is quite important, since a complex world model may take too long to build to be effective. There are too many states in a typical real-world environment to classify all parts of it. This requires the system to focus on what is important in deciding what to do and be able to ignore the rest of the details of the environment. How the world is represented and the areas of the world that are focused on to explore are key to determining the complexity of the task and how well the agent can accomplish it. A rational agent has many ways to learn (observing the environment, using a teacher, looking in books[15], etc.) and the agent may have to decide which to use in any given situation[16](14:4).

In choosing what properties of the environment are relevant, an agent sometimes must decide based on incomplete data. This *bias*[17] implies a tendency of choice when deciding between indistinguishable alternatives and is a part of any learning system that does not deal exclusively with 100% correct facts. The bias may be part of the language of the system, be it a programming language, a production system, or other representation. Or it may be inherent in the view of the environment, such as a fly is biased by its view of the world through faceted eyes or a robot only having auditory and collision sensors. In any event, the bias is needed to overcome lack of differentiation in the environment.

Additionally, a bias is needed to overcome noise in the environment which results in uncertainty in deciding what information was really presented to the system(14:7). Traditional learning approaches do not handle such uncertainty well, which can adversly effect how the agent responds in a noisy environment(14:7).

A bias is essential, however, to learning in a complex environment, since it is this bias that allows the selection between alternatives and the eventual generalization of the agent's models of the states of its world. A completely unbiased learner could never choose between alternatives and

[15]Books can be interpreted as look-up tables, etc., from the view of a computer-simulated agent

[16]This task is generally given to the usually human system designer.

[17]Which is effectively what we call an *inductive bias* later, since any decisions not based on given facts generally incur some form of generalisation or "guess."

so could never generalise its environment[18](14:6). Note that a bias might be inappropriate for a learning task, or just be completely wrong. This is one of the risks that the learner takes when confronted with an environment that must be learned. A good learning bias is one that allows the learner to be right enough of the time to effectively function in dealing with the presented environment.

In short, choosing a proper learning bias, be it by the designer, an outside agent, or by the system itself, is one of the most significant decisions made in a learning system, especially one such as an autonomous aircraft agent. Next we more formally define a "concept" as a means to represent the biased decisions made by the agent about its environment.

2.4.2 Concept learning. A *concept* can be described as a rule that divides the world into positive and negative examples(73:3). For example, the concept of "being red," as Schapire puts it, "... divides the world into those things that are red and those that are not red." The "learning" of the concept can be tested by presenting examples and seeing if the learner correctly can distinguish the new case. The "universe of objects" from which the learner is presented examples can be called the *domain* (or *instance space*) and each object in the domain can be called an *instance* of that space of alternatives. To use Schapire's example, if the domain is "all the fruit in the world," the learner's job is to distinguish red fruit from non-red fruit.

A prediction rule is one that allows the learner to categorise an example. Such a rule is called a *hypothesis* and is said to be *consistent* with the observed sample if it correctly classifies it into the right category. The examples presented to the system can come in some prescribed order, or they may be randomly selected, as might be the case in a typical environment. In many learning systems, it is important that the system be effective regardless of the order in which the examples are presented.

[18] One might argue that doing nothing is a decision (and a bias) in itself and so there cannot exist a system that has no bias in an uncertain world. That point, however, is not important, or denied by, the current argument.

One can measure the quality of a learning algorithm by looking at its expected performance on a test. More specifically, the *error* can be defined as the probability that an example will be misclassified. The *accuracy*, therefore, can be defined as the chance that the learner get's the classification right(73:5). Thus the goal is to reduce the error of the learning system to a minimal value, say, within some defined tolerance. We also desire the rate that the system reaches this value to be as fast as possible, i.e. the system should be *efficient*. This can be represented as a bound on the rate of learning (quickness that errors go down)(73:5).

The above model is called the *distribution-free* model, since the *target distribution* (the order that examples are chosen) has no effect on the accuracy of the system. This model is also called the *probably approximately correct (PAC)* model since the learning algorithm's hypotheses should be "... approximately correct (have low error) with high probability," and was first introduced by Valiant(73:6). A problem with this model, though, is its insistence on the error becoming arbitrarily small as learning progresses. Schapire makes the point that some acceptable value of accuracy might be sufficient: say 99.9%, or even 51%. Such a relaxed constraint would make the learning system a "weaker" system, but he argues (and shows) that such systems can be made arbitrarily "strong" by methods that improve the efficiency of any PAC-learning model. Thus "weak" learning systems, ones that are not required to (in the limit) become infinitely accurate, can be turned into nearly "strong" systems that come close to this goal(73:6).

To be more precise, "A class of concepts is *learnable* (or *strongly learnable*) if there exists a polynomial-time algorithm that achieves low error with high confidence for all concepts in the class. A weaker model of learnability, called *weak learnability*, drops the requirement that the learner be able to achieve arbitrarily high accuracy; a weak learning algorithm need only output a hypothesis that performs slightly better (by an inverse polynomial) than random guessing(73:13)." The two forms have been shown to be *not* equivalent when certain restrictions are placed on the instance space distribution (e.g. in the boolean function prediction problem the examples presented to the

system are uniformly and randomly distributed). But this restriction is not always necessary. The important point here is that specifying the distribution (order) of examples given the system, the efficiency of the learning algorithm is changed.

The *hypothesis boosting problem* is just that of "boosting" the low accuracy of a weak learning algorithm's hypotheses(73:14). By *filtering* the distribution of examples presented to the system, Schapire shows that hypothesis boosting can indeed be done and that a system that shows even 51% accuracy can be converted into a system that displays arbitrarily high (say, 99.9%) accuracy. This can be done by focusing the system's attention on the harder to learn parts of the distribution, thus taking advantage of the distribution-free nature of a learning model (assuming the model exhibits such a nature)(73:14, 18 – 51).

A *compression algorithm*, if it exists, can be used to reduce a data set into an equivalent but smaller one that represents the concepts to be learned(73:15,43). Thus, if such an algorithm does exist, it is possible to learn the related concept quicker by using the possibly much smaller reduced set of examples. In fact, for any given data set and concept, the number rules required to fully define the concept is independent of any sample size used to learn it[19]. Further, as Schapire shows, any learning algorithm can be converted into a compression scheme of this sort, which implies bounds on the complexity of the problem(73:15).

The idea of compression can be equated to the finding of the smallest logic circuit that represents a function. So, given a smaller circuit with a smaller number of inputs and outputs, it is possible to guess the function of the circuit with less tests. Of course, *getting* the reduced circuit is, itself, a problem, and an exponentially-bounded one at that. So Schapire's results are not as useful as they might be, since he assumes an *oracle* that can provide this selection of inputs. But the results do support the idea of speeding of the learning of a system by presenting the smallest

[19] A concept can only divide the examples in so many ways.

set of distinguishable inputs possible. This idea will be important when the domain of the problem is selected next.

Finally, we define *noise* to be variations of the sampled data set and *malicious noise* to be that noise that generates a misclassification of a sample(73:75). In a robust system, the goal is to effectively classify the input samples, even if they are a little noisy.

Note that the previous developments were for a two-valued system, such as one using data represented by the set $\{0,1\}^k$. Since any other form of representation can be converted to this form, it can be shown(73:41) that the results are likewise useable in these cases. Thus, if the input data set to the learning system is discrete and finite, it is possible to learn a concept related to the data in polynomial time.

2.5 Adaptive Search

This section looks at how we can define the autonomous agent problem as one of searching a rule space in an incremental fashion. A key idea here is that *all* agent problems can be addressed as learning problems, even though some only use rote or hardwired learning. First, standard search paradigms are reviewed. Next the problem is viewed as one of adaptive search, where the search space is adapted to the environment. This is similar to the idea of using context filtering as a means of speeding up concept learning. Then a search space and a solution space are constructed. Then search criteria are formalized.

2.5.1 Standard heuristic search. *Heuristics*(54:3) are "... criteria, methods, or principles for deciding which among several alternative courses of action promises to be the most effective in order to achieve some goal." In any search, heuristics make the decisions between alternatives that may have obvious differences in potential worth, or may have no apparent difference. A *good* heuristic can be defined as one that allows us to find what we are looking for in an efficient and timely manner. Thus the goal of the autonomous agent problem is to find a good heuristic that

allows us to search the space of all possible rules and find those that allow our system to perform in an appropriate way.

In a complex environment, finding an optimal set of rules becomes an exponentially-complex task as the number of possible combinations of rules that can be tried over a period of time in each environmental state grows. This is true even if we limit the tries to one rule per cycle[20] and assume that the number of possible states is relatively small. We can even assume that the environmental state is completely defined by the state representation posed to the search system, which is usually not the case[21]. Considering only these simplifications, the number of rules that link all possible states to all possible actions is still very large. Even with a static environment[22], finding the set of all such rules that best handles our agent's operations in this environment is, in general, computationally infeasible in complex[23] domains. Thus the need for heuristics to reduce the space of rules to search.

A heuristic search has four basic components(54:20):

- A symbol structure that represents the subsets of potential solutions (called *candidate solutions*).
- A set of operations (such as *production rules*) that modify the symbols in the structure to produce more refined potential solutions.
- A search procedure or *control strategy* that decides, at any given time, which operation to apply to the structure.
- A *state* representation that indicates what is left to search.

[20] For now, consider a *cycle* to be a discrete unit of simulation time. This will be elaborated on in Chapter 4.

[21] For example, looking at milk in a glass does not tell one if it is sour. Other information, not presented by the visual senses, is generally needed to tell that.

[22] A *static environment* is defined as one where, given a specific starting state, a specific action will always have the same effect.

[23] Complex in terms of the dimensionality of the input search space.

Search algorithms, such as depth-first search, A*, etc.(54) can be used effectively to find such rules in a static search space. The problem is that the environment (domain) we are searching for applicable rules changes as the search progresses. The heuristics used with these standard search techniques depend on this staticness to effectively exclude parts of the search space from the search that do not look promising. Since the static nature of the search space is only completely valid for a single time step, such a search would have to be repeated anv time the input domain significantly changed. Or, alternatively, a search of *all possible* contingencies can be done and supplied to the agent in rote fashion, which is the approach taken in many real-time applications[24]. But that approach precludes the agent adapting its search for rules dependent on the changing domain. For this reason we focus our attention away from static search methods and concentrate on adaptive search of the domain.

2.5.2 Adaptive search. John Holland, in his 1975 book ***Adaptation in Natural and Artificial Systems***(33) , showed how adaptive search can be used to speed up the process of concept learning by adapting the nature of the search as the search progresses. Here we formally define adaptive search and show how it essentially performs the data compression function needed to speed learning in an environment. Further, we look at how adaptive search can still be successful in an environment that is constantly changing.

An ***adaptive system*** is defined by the set of objects ($\mathcal{R}$, Ω, $\mathcal{I}$, τ) where(35:28)

R is the set of attainable structures.

Ω is the set of operators.

$\mathcal{I}$ is the set of possible inputs.

τ is the adaptive plan which, on the basis of the input and structure at time t, determines what operator to be applied at time t.

[24]Such as those that use a *universal plan*, as PDP-C does(31).

An *adaptive plan* is simply a function that maps inputs and a current search structure onto a new structure(35:23).

$$\tau(\mathcal{I}, \mathcal{R}) \rightarrow \mathcal{R}$$

τcan be considered as mapping $\mathcal{I}$ and $\mathcal{R}$ onto a set of operators in a nondeterministic way.

$$\tau(\mathcal{I}, \mathcal{R}) \rightarrow \Omega$$

where $\Omega = \{ \omega_1, \omega_2, \ldots \}$ and ω_i are operators.

We also define some other quantities:

$\mathcal{T}$ the set of feasible or possible plans.

$\mathcal{E}$ the range of possible environments (uncertainty).

$\mathcal{M}$ the memory of the system.

$\mathcal{X}$ the heuristics used to select a plan.

The autonomous agent problem can be formally framed as a problem in adaptive search. We define the search parameters as below.

$\mathcal{E}$ is the part in the environment that the agent is adapting to.

$\mathcal{I}$ is are the inputs received from the environment.

$\mathcal{R}$ is the structure undergoing adaptation. In classifier systems, these are the production rules themselves.

Ω are the mechanisms of adaptation. These are the genetic operators in genetic algorithms and the discovery operators in classifier systems.

$\mathcal{M}$ is what the agent remembers about the past. The message list and the rules themselves serve this function in classifier systems.

$\mathcal{T}$ are the limits to the adaptive process.

$\mathcal{X}$ describes how different (hypotheses about) adaptive processes are compared. Fitness is the form this takes in classifier systems.

By choosing an appropriate fitness measure, $\mu_E(\mathcal{R})$, to use to choose possible plans, the adaptive search can be guided into those areas of the domain that are the most promising(35:31).

Holland shows that such systems, using genetic operators as their mechanisms of adaptation, are robust systems that search out plans that produce exponentially better results(35:140). He also shows that such systems can be used to modify classifier-like structures such as in his proposed *broadcast language*, a precursor of classifier systems(35:153). Since the search plans are constantly evolving, they can deal with changing environments more efficiently that a fixed search plan. Indeed, by modifying the search procedure to select different sets of algorithms as inputs to the learning system, adaptive search performs a filtering function on the example space.

Adaptive search therefore is a form of concept filtering and this quality is desired in a robust adaptive search to speed it up, i.e. learning concepts by focusing on the differences in the environment to learn (what divides things into the desired concepts). The genetic algorithm applied to rule structures focuses on the differences between useful rules (those that receive payoff) and not useful rules (those that gravitate to low fitness). The fitness of the rules help to divide the rules into concepts. Other syntactic criteria may also be used, however, as in specificity of the rule conditions.

Holland also notes many areas of concern when designing an adaptive system including (1) the potential high cardinality of $\mathcal{R}$(which increases the search space), (2) apportionment of credit (determining which structures might yield above-average performance), (3) high dimensionality of μ_E(which can increase the difficulty of coming up with effective adaptive plans), (4) nonlinearity of μ_E(producing "false peaks" in performance), (5) search verses exploitation (dividing trials between exploiting structures that tend to give above-average performance verses creating new structures), and (6) how to use *payoff information* to better allocate trials. These are all issues

relevant to adaptive search systems to some degree, though not all may be applicable to any given problem(35:159-160).

2.6 Discussion and Direction

The goal of this chapter was to review the various literature addressing the theoretical aspects of the problem, and to lay some ground work for the next chapters. This section reviews what has been seen and offers some reasons for the direction taken in this research.

The chapter began by defining the various strategies that can be used in a learning system. These were applied to the autonomous agent problem with the result that all were shown to provide a useful function within the framework of the learning system. From here we presented some definitions that defined the problem as one of learning and as one of finding concepts (decision rules) that divide the state space into manageable generalisations. Then the problem was considered as an adaptive search problem.

A key idea that will be addressed later is that of concept filtering. It was shown that any learning algorithm's efficiency could be increased by filtering the inputs of the system to only those examples that represent interesting cases to the system. Whether represented as a concept filter, a search heuristic, or an adaptive plan, such input filtering is essential to allowing a discovery learning system with limited resources, and one that must process a limited number of examples before a classification of the inputs, to function efficiently within the target environment.

Further, since any state and any potential plan can be represented as an input to a concept (classification) system that in turn either accepts or rejects the plan based on the current state, decision systems can be made to choose plans of action in arbitrary complex situations. If these inputs are filtered to bring their complexity down to within the limits of the learning system's ability to efficiently process them, then such a concept learning system can, it is argued, handle the decision process in any autonomous aircraft agent situation. The problem then becomes one of

appropriate filtering of inputs to the system, instead of building a more complex learning system itself. Distributed agents that receive inputs and pass on filtered outputs to the next level can aid in reducing the learning system's search (rule) space complexity to a manageable level.

One problem not addressed so far is the need for correct filtering. If the examples chosen by the filters is noisy or incorrect, this effect can be amplified by the filters during the process of hypothesis boosting. I argue that this is a risk that must be taken by the autonomous aircraft agent control system and interfaces to keep the learning task complexity within the limits of the learning methods.

In the next chapter we review the various approaches that can be used to implement an autonomous aircraft agent controller. The focus is on how genetics-based adaptive search provides many of the different learning strategies in one system, making them potentially useful in our controller.

III. Examples in the Literature

This chapter discusses and analyzes the applicability of various relevant learning systems found in the literature. We examine various alternative solutions to the autonomous agent problem, focusing on the criteria developed earlier. These include systems using rote learning, deductive reasoning, and inductive search.

Rote systems and deductive systems start the review, and these systems have many of the qualities we identified (in the last chapter) as desirable for autonomous agent controllers, but not all. We also examine some other systems that show promise. These reviews are short, however, this they serve to lead us to the selected approach, that of genetics-based classifier systems.

Animats provide a fascinating view of the autonomous agent problem from a biological perspective. Animats are simulated life forms that inhabit a simulated world. It can be said that autonomous aircraft agents in a simulated airspace fit the definition of an Animat. Many animat systems have been implemented using genetics-based classifier systems. Also, many of the issues faced by animats are similar to those faced by autonomous aircraft agents in their simulated environment.

Classifier systems and genetics-based learning are then addressed, detailing the types of classifier systems available for use with this problem. These probabilistic systems use biologic-like operations and selective pressure to evolve rule systems that are adapted to the environment. These systems have been used in many contexts and have been shown successful in many applications similar to our aircraft agent control task in simulated, though small, environments. It is also shown that some enhancements to the basic classifier system will be needed to use the system in our domain.

The chapter ends with an analysis of these approaches, trading off the good and bad characteristics of each system. This analysis supports our selection of genetics-based classifier systems

to serve as the learning system for our controller. Chapter 4 takes off from here and shows how discovery learning can be implemented in these systems.

3.1 Rote Learning Approaches

3.1.1 Expert systems. Any expert system that interfaces to an outside environment can be considered a reacting autonomous agent. Most such systems employ rote learning by receiving and storing rules and attributes relevant to the domain they are designed for. Some systems, such as those built from the Kee expert system framework, also allow for the deduction of facts and operations based on forward and backward chaining of facts in the environment, as well as operations based on probability. These systems have been used to implement control and diagnostic systems. A control system implemented in Soar is one example (discussed in a later section).

These systems are still slow, and generally have limited reasoning capability in real-time systems due to the overhead of the reasoning processes of the architecture. Research is looking at the possibilities of speeding up these systems, and many show promise for future use in complex autonomous agents. Since these systems are outside the current scope, however, they will not be considered outside of this chapter.

3.1.2 Pilot Decision Phases in Clips. The Pilot Decision Phases in Clips (PDP-C) system is a rule-based system implemented in Clips COOL (Clips Object-Oriented Language)(31, 32). It provides for the simulation of multiple agents in a near real-time environment[1].

This is a reactionary system where all possible situations have been encoded as rules into the system and the agents progress from phase to phase in a deterministic manner. Though complex

[1]The system is based on rule firings, and complex situations can lead to numerous rule firings that slow the system down. Since its execution varies and is sometimes relatively slow, the system is not considered a real-time system. Note that by tracing all possible execution paths and determining the maximum execution time of a cycle, and by ensuring that the system (via sufficient processing capability) can execute the longest execution path within the time of one cycle, the system can be reinterpreted as a real-time system.

behaviors are shown, they are not adaptive and cannot deal with situations not previously designed into the system.

The *universal plan* approach is an effort to encode all possible situations along with plans of reaction into a rule base that can be used by systems such as PDP-C. Though such a rule base can never be complete in a world with infinite variables, it can provide a relatively good set of behaviors to the agent. The goal is to continuously improve the viability of the plan by enhancing it whenever weaknesses surface in simulations that use it.

A question one can ask is, "Will the limitations of the simulation environments used for testing the system make the rule base 'brittle', i.e. overly dependent on its environment for the plans to be useful in all situations intended?" The addition of some form of on-line reasoning to cope with and learn from new situations seems to be one approach to addressing this potential deficiency.

3.2 Systems Using Other Learning Methods

This section quickly reviews other systems related to the autonomous aircraft agent problem. The purpose of this section is to give the reader some idea as to what other approaches can be tried. Since we have narrowed the scope of this investigation to using genetics-based classifier systems, no comprehensive comparisons on applicability of these systems as compared to our selected approach is made.

The first architecture reviewed is Soar, a software system designed to be a general architecture for artificial intelligence. Next, the Pilot Associate effort is examined, which, although intended as an assistant to a pilot, has many of the qualities of an autonomous aircraft agent. Last, efforts with MAVERICK, a discovery-based system related to routing in the aircraft agent domain, is examined.

3.2.1 Soar. Soar is a software architecture for general intelligence that has its roots in the General Problem Solver (GPS) efforts of the 1960s, from which it is a direct descendent(67). Soar has grown and evolved over the years, with new, more capable versions emerging every couple years: Soar (Soar1) in 1982, Soar2 in 1983, Soar4 in 1986, Soar5 in 1989, and Soar6 in 1992. All versions up to Soar5 were written in LISP, while the latest, Soar6, is written in the "C" programming language for efficiency and portability. Early versions of Soar used the Xaps2 production system, while all versions from Soar2 on are based on the OPS5 production system approach. Soar is an architecture that can be used to implement other systems (such as Neomycin-Soar) that require a general goal-based reasoning mechanism.

Soar is an architecture for implementing reasoning systems that uses chunking, a basic grouping mechanism (cf. chapter 1). Soar focuses exclusively on a ***problem space***, i.e. a space with a set of operators that are applied to a current state to yield a new state. Thus, all tasks in Soar take the form of heuristic search. In fact, Soar uses this problem space as the fundamental organization for *all* of its goal-oriented symbolic activity, which is based fully on the ***Problem Space Hypothesis***, and all decisions in Soar relate to searching a problem space (selection of operators, selection of states, etc.).(43:467-468)

Soar approaches a problem as the process of searching for a way to accomplish a goal. When, in the process of searching, Soar discovers that a particular part of the task cannot immediately be accomplished, it sets up a ***subgoal*** to find a way to meet this need. When all such generated subgoals (termed ***impasses***(43:471)) related to a goal are finally addressed, Soar then has the knowledge it needs to continue achieving the original goal, and so it does. This ***automatic subgoaling*** is built into the Soar architecture and provides the basis for many other of its features.

Soar builds knowledge ("learns") by various methods. The "weak methods" of hill climbing search, means-ends analysis, etc. are implemented via search control productions. Soar exhibits these search actions as part of its architecture without any need to explicitly specify which to use

in any given case (given that the operators defined for a problem allow states to be evaluated in an applicable way to such a search). The primary learning mechanism in Soar, however, is *chunking*(43:472). When a solution to a subgoal problem is arrived at, this solution is formed into a "chunk" of knowledge that can be applied as a unit whenever a similar situation is encountered. This process, along with the subgoaling feature of Soar, allows Soar to decompose and learn the pieces of complex tasks.

"[Soar] is a system intended to exhibit general intelligent behavior(71)." By using search in problem spaces governed by a fixed decision procedure, and only using a rule-based production system for long-term storage, the authors of Soar argue that knowledge relevant to solving a problem can be called upon to supplement the search by merely matching the appropriate rule conditions in the long-term rule base. By using the OPS5 and its underlying Rete matching algorithm, the system efficiently finds solutions to the problems it receives as goals to achieve. Nonetheless, the Soar system carries a significant overhead and may not be appropriate for real-time problem solving.

One area where Soar has been shown effective is in explanation-based learning (EBL), incorporating rote memorization and multi-example induction(66). Here Soar is able to generalize examples to a degree and provide chunks that are usable in other, related contexts. This generalization uses the omission of various parts of rules learned to build (via a form of deduction) rules that apply to more situations than the original rules were learned in. This process is heavily dependent on the examples provided to the system, however.

Soar has, among other applications, been shown to be able to control a robot in real time(44). Called Robo-Soar, the system controls a Puma robot arm using a camera vision system to guide it. It uses an abstract look-ahead planning strategy where planning results by creating an internal model of the environment (via productions) and applying a sequence of operators to the model to see if failure is predicted. Planning is used to build up the plan to solve a new problem or

subproblem, then this and any other stored plans can be used to solve similar problems, or parts of similar problems.

The work at AFIT(29) has considered using the Soar approach to learn (find via Soar's search mechanisms) better fighter tactics to employ in the *universal plan*, a database of tactical rules. By supplying Soar with examples of tactics that have shown utility in the fighter simulation environment, the hope is that the Soar architecture will, via the mechanisms discussed above, find better rules that cover a larger range of situations. Other simulation systems could then test these new rules in simulated scenarios that can include simulated combat between autonomous agents. If agents using the new rules outperform other agents in the system, then those rules are considered for inclusion in later versions of aircraft agents.

3.2.2 Pilot Associate. The Pilot's Associate project(2), a joint Defense Advanced Research Projects Agency (DARPA)/USAF program, was involved with creating an "associate" system that can project the pilot's needs in all situations and provide guidance and assistance. The interest here is that the system could also, if allowed, take on some of the lesser tasks involved with operating a fighter craft, allowing the pilot to concentrate on the overall picture of the mission at hand.

The system addresses many of the pilot's needs as planning issues. This involves the projection and selection of resources and actions to accomplish each aspect of the mission. These operations can be considered low level to a pilot who must assimilate all relevant details and react on a moment's notice. To this end, the Pilot's Associate focuses on "chunking" (in a similar way that Soar (cf. previous section) does) as a means of "packaging" information into higher-level "chunks" that the pilot can understand and use more readily. The key is to give the pilot a set of useful, relevant choices at the moment they are needed. The effect is to put back (electronically) the back seater that has been removed from the latest fighter aircraft(30).

Returning to the focus of this discussion, the architecture used by the Pilot's Associate is said to in many ways mimics that of the pilot. Indeed, it would have to if it is to anticipate the pilot's needs and present relevant suggestions for action. It is this ability to guide and control the systems of the aircraft that make it of interest in the simulation of autonomous agents.

The nature of the system (a hybrid of various reasoning subsystems connected via a communications network) parallels in many ways the distributed architecture approach we propose. The complexity of the subsystems, however, and the limitations of processing hardware at the time of testing limited the implementation of a complete aircraft control system. This system is outside the current design scope, however. The reader is referred to the literature(30, 2).

9.2.3 PAGODA. The PAGODA (Probabilistic Autonomous GOal-Directed Agent) learning system by desJardins demonstrates how goal-directed learning can be applied to the search for efficient, on-line behaviors(14). The PAGODA system operatee in a simulated robot domain called RALPH Rational Agent with Limited Performance Hardware)[2](14:2). The agent in the simulated world has the primary task of maximising expected utility. This is done by using a model of the world to make predictions about it a fixed number of steps into the future. This forward-chaining look-ahead allows the system to make a best guess as to the results of executing a specific action. Note that RALPH is a rather complex world[3] which impacts the complexity of the learning task.

The system distinguishes between *learning goals* (goals that facilitate learning intermediate steps to a goal) and *planning goals* (final goals that the planner deals with). In other words, "A learning goal is a feature of the world which the agent's inductive mechanism builds a model to predict(14:50)." The system needs to be able to predict these intermediate states, to determine the overall outcome of a particular plan at reaching a desired planning goal. In a way similar to

[2]The RALPH simulation system, which runs on a TI Explorer in ZetaLisp and on DECstations in Allegro Common Lisp, is an object-oriented system with scheduling software and a graphics display. The system is available by sending email to ralph@guard.berkeley.edu(14:28)

[3]Similar in many respects to those used for animats. In fact, this application qualifies as an animat problem as defined by Wilson (cf. later in this chapter).

classifiers, the system stores an expected utility per unit time for each such goal and uses this in the forward search for the set of actions with the highest end utility[4]. Intermediate states are assumed to have the same utility as the end goal state.

A *single-step plan* is defined as a triple of form (a, pw, u), where a is an action, pw is a perceived world, and u is the expected utility of taking action a in world pw. Using this single-step model, a jump to the indicated state via the given action will have the given utility. The plan that achieves this state in the world may be composed of smaller feature goals. The choice of what to do is based on a weighted look at the utility of the previously tried rules that make up a plan (rules that haven't been tried are assumed to have zero utility, i.e. neither positive or negative)(14:56).

Empirical tests using PAGODA in the RALPH environment show that learning is taking place, but that it is not perfectly accurate(14:123), which is expected to be due to the complexity of the domain. The system is probabilistically based and not as much heuristic search, so the implementation is not as efficient as it could be. Also, without pruning, the simplistic forward-chaining search is noted as inefficient and thus another area of further research. The main point made by desJardins is the need for methods to constrain learning: agents must learn to learn better(14:153).

3.2.4 MAVERICK. Freeman Kilpatrick's MAVERICK(39) is discovery-based learning system that learns maneuvers in a route-planning domain, and was developed by him during his graduate study at the Air Force Institute of Technology. The aspects of this work of interest to this investigation include the implementation of the discovery learning mechanism and the particulars of the domain, which are related to the autonomous aircraft agent problem.

[4]Actually, the equations used are more complex and take into account the costs of planning, of time spent experimenting, etc. The net utility of learning a goal is then a function of expected costs and utilities that is not typically linear. PAGODA simplifies the task of utility assessment by only looking at the change in utility the results when a particular learning goal is achieved and ignoring the costs.

The task was chosen to be that of planning a route around a set of threat objects (surface-to-air missile (SAM) sites, aircraft, etc.), minimising detection by threats while providing the most time-efficient route possible. Kilpatrick's system, written in Common Lisp, uses RIZSIM, a general purpose combat environment simulator (written in "C"(39:3-4)) developed for parallel processing research at AFIT by Robert Rizza(39:3-5).

Kilpatrick notes that an *agenda* is central to any discovery-based learning approach. Such an agenda is used to store the tasks to be performed, as well as their interestingness(39:2-7). An agenda is useful to guide the search for better solutions (to the routing problem in this case) to avoid wasting time in areas of the search space that are less fruitful. In essence, the agenda provides the input filtering to the system.

The concepts to be learned by the system are variable-length sets of maneuvers composed of a variable-length series of turns. Thus a solution is a set of maneuvers, each of which is a set of turns. So, even though a route will be generated by executing the derived set of maneuvers, the search is actually for these maneuver strings. The search space then is the set of all possible maneuvers for a given scenario(39:3-6).

The system used a set of heuristics to control the generation of new maneuvers and another heuristic to arrange these potential solutions on the agenda list. Maneuvers started with the NULL (turn-less) maneuver and turns were added to generate the various routes to try.

The implementation used a set of files to pipe the various flows of information between the various components. A scenario file was loaded into the RIZSIM simulation, which produced an output file read by MAVERICK. MAVERICK, in turn, generated a maneuver file that RIZSIM received and processed as well as history files that documented activities.

RIZSIM objects are modeled as generic moving objects that have an initial route and starting position and velocity vector(39:4-2). A problem with this simulation is that objects can't stand still and still exist. Kilpatrick handles SAM sites by moving them at a very slow rate. Also,

the interactive ability of the simulation is lacking due to a use of data files to input and output route data to and from the simulation. This interface could be enhanced, in theory, to support a more interactive approach. Many other object attributes are available in the simulation, but not considered here.

The basic MAVERICK system uses a learning structure that stores *interestingness* (an integer assessment of utility), *move* (a set of sequenced maneuvers), *age* (a sequence of maneuver in total number of maneuvers explored), *heuristics* (a list of the heuristics used to form this maneuver), *level* (the tree-depth of the maneuver), *man-time* (maneuver time in seconds), *total-radar-contact* (the total radar contact time (from SAM sites) of the maneuver), *radar contact* (a list of individual radar contact times per simulation object), *radar-directions* (where the radar came from: L (left) or R (right)), *exec-time* (CPU time to test this maneuver), *child* (a list of children of the maneuver), and *parent* (the parent of this maneuver). The agenda uses the interestingness to select which maneuver to test next(39:4-11).

The search technique employed is mainly heuristic-guided mutation. Kilpatrick notes that hill climbing is one tendency of such a technique and uses various mechanisms to assist the search, including maneuver aging (older best maneuvers, which are the best of a family of mutated maneuvers, are more likely to be near local optima and so are less likely to be searched further), selection and transformation heuristics, and ordering heuristics (the order to apply transformation heuristics). A *scenario memory* as well as a longer-term memory are also used to save useful maneuvers so they can be reapplied at other times in the hope they provide a short cut to different situations in the simulation(39:419).

Kilpatrick noted some problematic tendencies when MAVERICK is applied to the multiple SAM site routing problem. Since the scenario memory stores potential sub-maneuvers to be tried for use on other objects in the simulation, a large number of such potential solutions can be generated and slow the discovery process. Also, since only straight-line radar coverage (the radar coverage if

no maneuver is taken) is used to index past long-term memory solutions, the applicability of the stored solutions varied(39:5-12).

His exploration of the various contributions made by the different heuristics showed that the learning system could compensate for the loss of one or more guiding heuristics, but that such a loss increased the number of maneuvers checked(39:5-17). This shows promise for the discovery learning approach in general.

Overall, Kilpatrick demonstrated many of the concepts of discovery-based learning techniques. Though the system is limited in the ways described above, these techniques can be applied to other systems to enhance their performance.

3.3 Animats

Animats are artificial animals that interract with simulated environments(80). Animats have gained notoriety in the computer science, machine learning, ethology (animal behavioral science), and genetics communities because they provide a method to test theories on behavior, intelligence, and evolution within the framework of a controllable simulation. Of much interest is the study of adaptive behavior in animats as they react to the sensory inputs from the artificial environment they find themselves in. This section first reviews what makes up an animat. Then a short overview of animat research is presented with emphasis on the adaptive nature of these systems. Finally, we examine how animats address some of the needs of the autonomous agent problem.

3.3.1 What are animats? ***Animats*** are adaptive artificial life forms that inhabit simulated environments. The key aspects are ***adapting***, i.e. the animat must be able to continue to function ("survive") in the face of a changing environment, and ***changing***, meaning that, as in any real environment, change is inevitable and must be dealt with by the animat. These qualities appear in

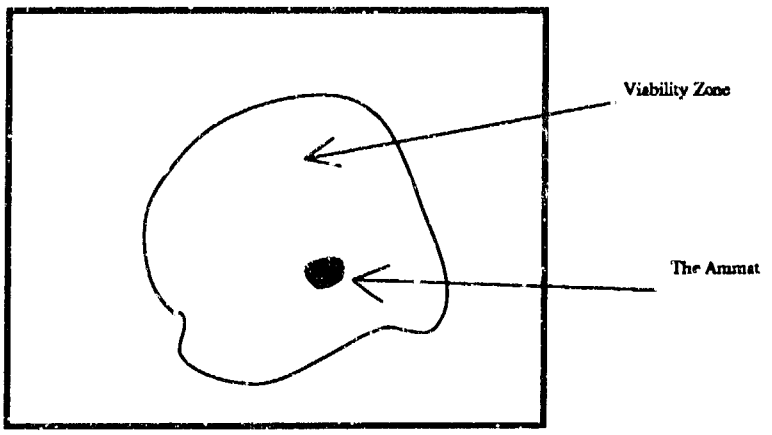


Figure 3.1 Viability zone of an animat.

greater or lesser degree depending on the application[5] that is being implemented. In the robotics arena, for instance, an animat becomes an autonomous robot that interacts with a real, albeit controlled, environment (the floor of the robotics lab, for example). What counts is the interacting of some agent with some usually simple environment in an autonomous and typically adaptive way.

The animat problem is defined by a number of *state variables* that describe the animat's total environment[6](48). The job of the artificial life form is to keep these so called *essential variables* (introduced by Ashby in 1952) within ranges that allow the animat to continue surviving. This set of ranges defines a *viability zone* inside a given state space, and the animat can be referenced as being at a specific point in this space at any given time[7]. The animat can be considered *adaptive* if if can determine where the borders of this region are and avoid transgressing them(48:2). See figure 3.1[8].

There are many ways to produce this behavior in an animat. For instance, the artificial animal could rely on various homeostatic mechanisms that tend to return the organism to the middle of

[5]Remember that we have defined (back in Chapter 1) the term *application* to mean an agent in a particular environment that it must interact with to perform some function or task.

[6]For simplicity, what the animat can sense determines its environment, although other unseen forces may be at work. The filtering concept presented earlier is accomplished outside the scope of the model.

[7]To give a natural analogy, this is equivalent to a reindeer in the artic avoiding thin ice that might break. The deer must also avoid situations where a predator might get it. All such lethal situations exist outside the viability zone and so define it.

[8]From the figure in Meyer and Guillot 91(48).

the viability zone if it strays too far from it. Alternatively, some form of sensor mechanism could warn the animat of its approach to such a lethal border and allow it to react accordingly. Such actions might be *reflex* (direct reactions to input stimuli, such as happens in the well known child and hot stove example) or may involve a more deliberated chain of reasonings. But such reactions are generally fast and of an "instinctive" (pre-programmed) nature. Meyer and Guillot note that animats that can react beyond the stimulus-response level have the capability to choose the form of reaction, which can result in better survival potential for the animat(48).

Other skills of use to the animat are a form of memory that can record sequences and relationships encountered and their utility, as well as some form of planning[9]. Both enhance the agent's ability to handle more complex stimulus, such as circumventing a set of obstacles in an efficient manner. These and others are discussed next.

Wilson provides a good introduction to the animat approach in *The Animat Path to AI*(81). He also makes the point that one path to more "intelligent" systems is to start small and work up. Animats, he argues, are good for studying animal behaviors that can be built upon to eventually reach human levels. Of course this does not address the complexity issue that follows all attempts at increasing the domain space of concept learning systems, which all learning systems, including animats, are versions of. But the approach can still provide insights into behavior that may provide better heuristics for filtering and categorizing the large volumes of input humans deal with.

3.3.2 A short overview of animat research. We now take a quick look at the diversity of the research on animats. To a good degree, these areas parallel work in other related fields already discussed, as well as in the next section on *classifier systems*, except that animats have deeper roots in the ethological and biological sciences. Note that this historical review is mainly from *Simulation of Adaptive Behavior in Animates: Review and Prospect* by Meyer and Guillot(48) – a good review the reader is referred to.

[9] As was suggested by desJardins earlier in this chapter.

One can divide the behaviors of animats into three different categories: preprogrammed, learned, and evolved. Each of these will be looked at in turn.

Preprogrammed behavior

Based primarily on rote and implanted learning as well as "hardwired" feedback loops and other "non-reasoning" mechanisms, these are the techniques used in some of the earliest animats. In fact, the only difference between these systems and computer programs (in a learning sense) is the animat's need for and existence in an environment. Booth's feeding behavior model of rats (1978), for instance, is based on the calculated energy flow in to and out of the animal's tissues. "Hunger" is a result of a hysteresis loop that governs this energy flow(48). Likewise, another cybernetic model of drinking behavior proposed by Toates & Oatley (1970) also use a similar feedback loop(48).

Extending this approach, many researchers are trying to increase the adaptiveness of animats by providing them with more and more realistic sensory inputs based on those of animals. Examples include vision based on a fly's compound eye and a toads visual system, the dolphin's echolocation system being applied to neural networks, and various studies of motion coordination in animals(48, 79, 65, 47). An issue that emerges here as elsewhere is that of ***credit assignment***, i.e. determining what actions should be credited with the success or failure of a previous situation. This is especially important when the *payoff* (perception of success or failure by the organism) comes at intermittant times[10]

Deitic representations provide context filtering in that only the sensory inputs relevant to a task are provided the animat (Agre 1988)(48). An example might be a student studying a textbook at the exclusion of other noises and images around him. This approach also gives rise to systems that actively control their sensory inputs, an approach we will consider later in more detail.

[10]The credit assignment problem is addressed in more detail in the next section on classifier systems, though it is a problem faced by any system that is not told (directly or indirectly) the utility of an action when it is executed. Indeed, this is one of the things an inductive learning system generally needs to learn.

Work with artificial neural networks and other means to simulating natural nervous systems continues, supported by work in the field of ***neuroethology*** (Ewert 1980, 1987; Camhi 1984)(48:3). Meyer and Guillot cite one example of an artificial insect built on a neural network model that displayed many fly-like behaviors (wandering, edge following, feeding, etc.) showing the simulation of a sensory-motor system in an animat. One problem to be avoided in such systems is the conflict when many possible but incompatible actions are requested at once (looking for new food versus eating the food at hand). This is generally avoided in programmed animats by implementing a hierarchy of actions where one action (say, eating) takes precedence over another (say, flying in search of more food). This also highlights the point of ***default behaviors*** or behaviors that are followed when no more pressing need exists (e.g. random exploration)(48).

Robots based on the programmed behavior approach generally depend on detailed models of the environment they interact in. Robots also can use ***motor schemas***, basic units of motor control, to build more complex reactions(48:4).

The ***subsumption architecture*** of Brooks (1986) is an approach that does not need such detailed world model(48). This architecture wires sensors more directly to action-suggesting modules instead of an internal model. A program is written by specifying layers of networks of finite-state machines augmented with various timers and registers. Such approaches allow for the creation of creatures like Squirt (Flynn, *et al.*), a small robotic cube that hides in dark corners and ventures out only when noises go away(48).

Cooperative agents can be used to build interactive "societies" of animats that work to achieve some task. These can be tightly coupled into some system, such as in modeling the various senses and actions of a rabbit, or be distributed and loosely coupled, such as when a group of robots are sent to the moon to work at building a lunar landing site(48). Another example are modeling insect societies (Moyson & Manderick 1988; Steels 1987, 1989)(48).

Learned behaviors

Learning behaviors in animats takes on many forms. Simplest is *conditioning* where the temporal relationships of events are used to build a *temporal difference model* of what to expect when the environment reaches a certain definable state (in terms of a concept being learned that triggers on a set of equivalent states)[11]. Another distintion is that of *conditioned stimulus* (where the learner is trained to react to it) versus *unconditioned stimulus* (where the learner reacts "instinctively"). The later can be considered the "hard-wired" rules of the system, while the former the learned relationships.

Supervised learning has been used to train the NAVLAB autonomous vehicle of Carnegie Mellon(48). *Unsupervised* or *discovery learning* has been used in many examples to train animats in simple environments. Related to this is *reinforcement learning* where a gain (payoff) or signal is generated externally whenever the animat performs "correctly".

An animat, as it learns its environment, may build a *cognitive map* of the environment that is used to direct later actions. Being able to maintain a model of the environment internally allow an animat to draw upon this model during times of intermittant feedback from the environment. Such a model also helps to alleviate the credit assignment problem that such intermittant payoffs generate.

An architecture that has an internal model of the world that it uses to predict real-world actions is the DYNA architecture (Sutton 1990)(48). The DYNA architecture has four basic structures: a real world it interacts with and provides rewards and punishments, a world model that is intended to mimic the real world in a one-step fashion, a policy function that chooses the animat's actions, and an evaluation function it uses to judge the worth of a course of action. The goal is to maximize long-term average reward.

In the DYNA model, the animat updates its world model, policy function, and evalutation function based on interaction with the real world. Experiments on the real world produces hard

[11] Thus teaching a system a task is similar to the conditioning techniques B. F. Skinner used on pigeons to teach them to perform complex behaviors to receive food at the end.

data used in *temporal difference reinforcement learning* and experiments on the internal world model update the system by *relaxation planning*. The algorithm used is related to dynamic programming and uses a credit allocation scheme similar to the bucket brigade. Considerable success in some domains has been shown(48).

Another system, called AGAR (Travers 1989), is based on Minsky's *Society of Mind* work and uses *agents* with condition/action interfaces to perform specific functions in a system when triggered by sensor inputs or other agents(48:6). Actions are encoded in LISP and so can be nearly anything programmable. Travers has applied this model to a biological fish mating context.

Still other work has studied the use of the *deitic representation paradigm* mentioned earlier to not only limit the inputs to the animat, but let the animat learn where best to focus attention. This approach is a significant one in that such representations provide a filtering of inputs received by the learning system based on time. Though not mentioned in this source, such a filtering could also be used to limit the available actions the system must know about in any situation. Related to this is the problem of *perceptual aliasing*, where the internal states of the animat's world model don't distinguish between two external states (Whitehead & Ballard 1990)(48). This can result in any system that filters inputs to another representation for use by another system.

Finally, *classifier systems* fall into this category, but are covered in the next section.

Evolved behaviors

Evolved behaviors come about by selective pressures in the environment that force the system to adapt over time. Adaptation can be the modifying of parameters that control the activities of the animat, or they might come about via the use of encoding characteristics into genotypic structures that undergo modifications under selective pressure. Parameter evolving is addressed briefly at the end of Appendix A. The later is addressed more in the next section.

Other examples of evolutionary animats abound. For example Bertin's (1990) study of aquatic animats called *paddlers* that seek out and eat *glowballs*(48:8). This is an evolutionary system built on a neural network that has its parameters mutated to facilitate adaptation.

3.4 Classifier Systems

This section generally describes *classifier systems*, rule-based learning systems that can adapt to their environment, and shows how they provide one alternative to the other machine learning architectures so far presented.

3.4.1 Overview of classifier systems. This section gives a brief look at the makeup of the "standard" classifier systems. They generally use *genetic algorithms* as the primary means of rule discovery. For an introduction to genetic algorithms and how they are used in classifiers and other machine learning approaches see Appendix A. For a more detailed introduction to classifier systems see Appendix B. For a short review of classifier system development see Appendix C. Chapter IV builds from this material as it details the theory behind how discovery learning is added to the PPLS learning system design.

The first system, known as the standard or "Michigan system" (see below), is build on the structure shown below in figure 3.2[12]: This system receives inputs from a set of *detectors* that monitor various attributes of the environment. These inputs are encoded into *messages* that are put on a *message list* on any given processing cycle. Then, that same cycle, the messages are matched against the *conditions* fields of *classifiers* (rules) on a *classifier list*. Those rules that have their conditions matched compete to activate their *action part* that, in turn, generate either commands to awaiting *effectors* that change something in the environment, or new messages for other classifiers that go on to the message list the next cycle. Old messages (except for the new action-generated messages) are then purged and the cycle repeats. New rules are added by running

[12]This system is discussed in depth in Chapter 3.

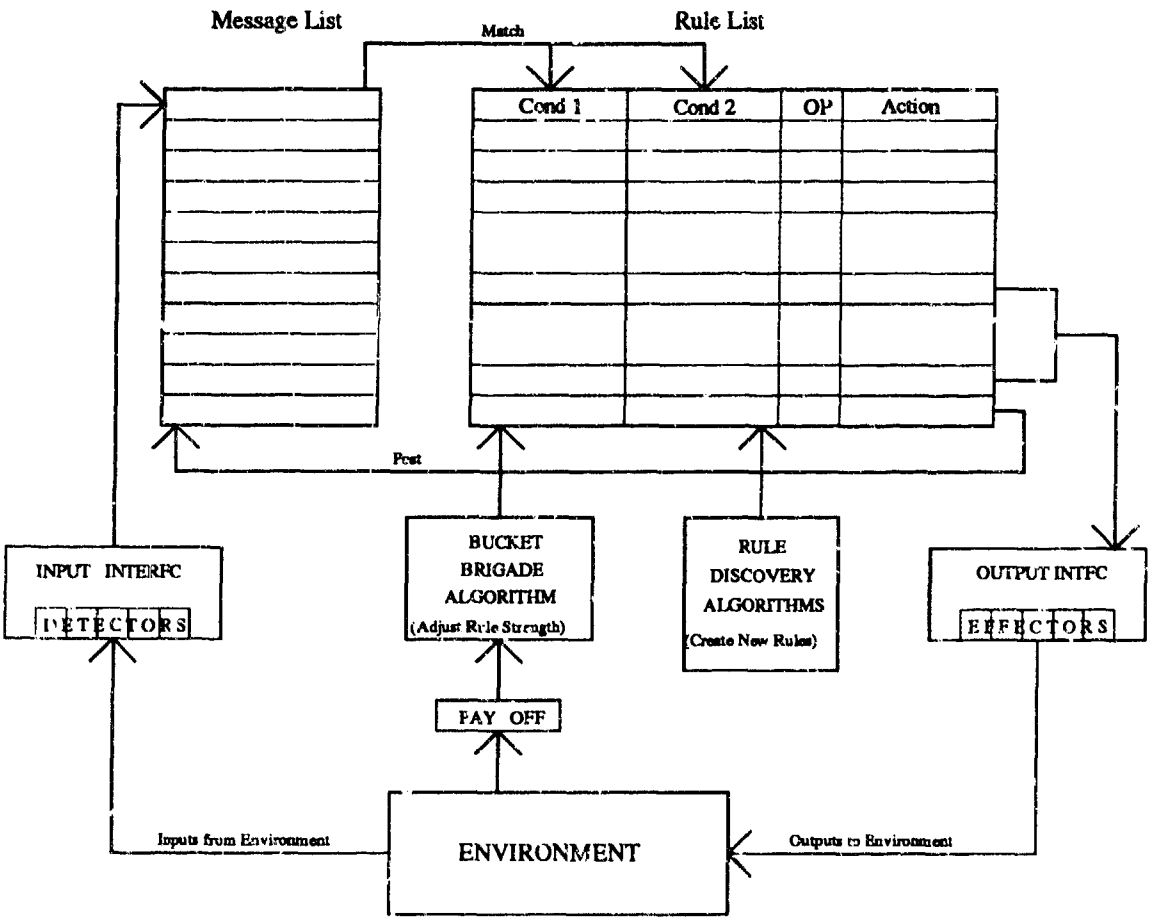


Figure 3.2 The standard or "Michigan" classifier system.

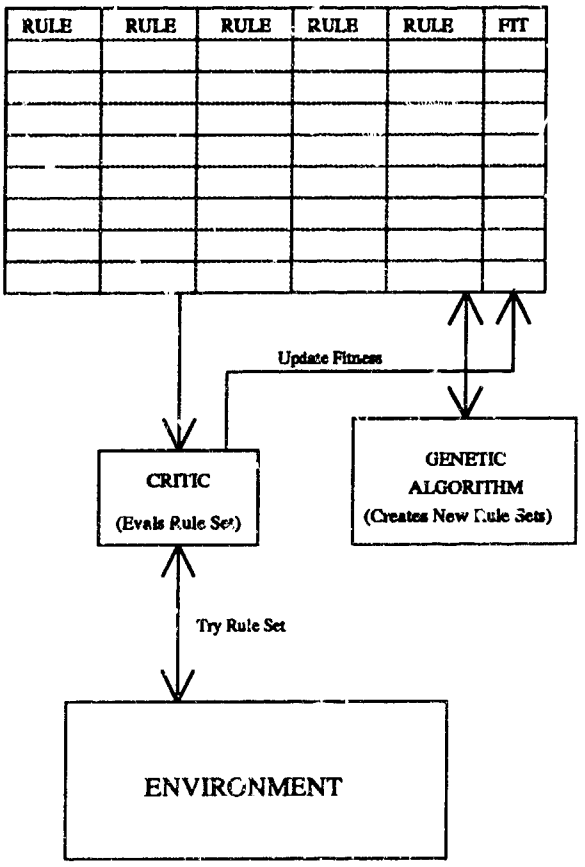


Figure 3.3 A "Pitt" classifier system.

a genetic algorithm[13] and other rule generation algorithms on the rule population (i.e. classifier list).

An alternative approach, the rule set or "Pittsburgh" approach, functions more like a rule-processing standard genetic algorithm than a standard classifier system. This system, shown in Figure 3.3, uses genotypes that encode entire rule sets into each population member. Each member is then tested on a set of test cases and given a fitness based on how the entire rule set did. In the simple case, the contribution of each rule is not judged. After enough trials, rule sets that are optimized to the task can emerge.

[13] See Appendix A for an introduction to genetic algorithms.

There are many arguments about which system is better, and each have their own worries. These are addressed in Appendix C. The rest of this chapter focuses on "Michigan" systems, since this is the approach selected for the learning system implementation.

3.4.2 Notable Classifier System work. In the last four years or so many innovations have appeared in hopes of making the better classifier. A sampling of these approaches is given here, including Booker's Gofer system, Grefenstette's Samuel system, Riolo's CFS-C classifier package. Some other systems of note to this research are then listed to show other important directions in the research. Each of these systems have qualities desirable for our autonomous aircraft agent classifier application.

Gofer

We start this rundown of interesting systems (from the view of this investigation) by looking at Lashon Booker's Gofer system. Many of the innovations here are offshoots of his 1982 dissertation and later work. We look at both here.

As mentioned previously, the key ideas proposed by Booker are the modification of the matching algorithm to allow more freedom as to what matches the input messages of the system and the limiting of the population during genetic operations to those rules that have been recently matched. These two modifications force the rule discovery algorithms to focus on schema that have shown some ability to address the conditions that the learning system is currently dealing with since the active rules have already been narrowed down to those that have applicable condition fields. Note that the cover operators addressed by Wilson and Riolo essentially generate rules that fill in these gaps, though with randomly selected actions.

By trying to find rules that specifically meet the requirements of the input detector state, the system is filling a *niche* in the population that address a specific concept[14]. This cluster of rules

[14] One assumption here is that all rules that are active are in fact filling the same need. If the rules are general enough, they may be trying to fill a niche that spans multiple concepts. Evolutionary pressures are generally assumed to force the system to evolve the subniches necessary to let the system learn these concepts correctly.

that fill a niche addressing a concept is called a *species* by Booker and is the basic grouping in these systems(4).

Forming niched subpopulations is similar to the multimodal optimisation that DeJong observed using genetic algorithms in his 1975 disseration work(13). The goal is to build multiple subpopulations of rules that each address a specific niche in the environment. Even with Booker's modifications, however, learning in such systems is not easy if the search space contains multiple dimensions and multiple concepts to be learned. His approach starts to fail drastically once the complexity of the domain space or action space exceeds some application-dependent value(4). Also, his work only concentrated on simple stimulus-response rules. It has not been shown how speciation can support the building of action sequence chains to address the handling of sequences of related environmental events.

Nonetheless, niching is a useful tool and will be readdressed in Chapter 3 when the design of an autonomous agent system is looked at in more detail.

Samuel

Grefenstette in 1991 looked at using Lamarckian learning in multi-agent environments(28). Lamarckian learning allows the passing of environmentally useful characteristics and behaviors to later generations. This contrasts with Darwin's theories where only the genetic makeup is passed to the offspring, undergoing chance variation in the process, and natural selection determines the characteristics that survive. However, as Grefenstette notes, the restrictions on natural systems need not apply to artificial systems. The described research uses Grefenstette's Samuel system, which has been the focus of continuing efforts in classifier research at the Navy Center for Applied Research in Artificial Intelligence.

The system works by passing not only the rules from one generation to the next, but also the strength of the rule, allowing past performance of the rule to influence the selection of rules later[15]. A *bid bias* is also introduced that allows the strength to be varied:

$$\text{bid} = \text{strength}^{bidbias}$$

If the bid bias is zero, then the strengths are ignored during competition (if they are greater than zero). If between 0 and 1, then lesser-strength rules get a boost in the competition. When the bid bias is 1, then the bids are as in standard classifiers. Greater than 1 and high-strength rules are favored. Grefenstette implements a maximum fitness value, so a really high bias (> 10) biases all rules with strength greater than 1 (the average) at the maximum value. This last setting seems to give the best results in the examples studied.

Another attribute of Grefenstette's Samuel system is the use of various *learning operators* that allow greater variety in the types of matches made. The operators used in this system can be decomposed into four basic types: linear, cyclic, structured, and pattern.

Linear and cyclic attributes (condition fields) take on the form of linear or cyclic progressions of numbers. For instance, a field defined as(28:305)

```
(time is [5 .. 10])
```

matches any number from 5 to 10 in that field. Cyclic values act in a modulus way so that

```
(direction is [270 .. 90])
```

matches any value from 270 to 360, and from 0 to 90.

A pattern attribute is the traditional condition representation

```
(visual-field is 0###1)
```

[15]This is the typical case in most classifier systems, so most systems, using the "Michigan" approach, are thus Lamarckian. Note, however, that Grefenstette's system is "Pitt"-based and "Pitt" systems, by default, don't use individual rule strengths.

that specifies a bit pattern to match (with # being a wildcard matching either 0 or 1). Structured values are basically an enumeration type where the field can match one of the listed values.

```
(distance is [close, 400])
```

The structure is that of a tree of values, so, if the value of "close" included "very-close" and "medium-close", and these included {100, 200} and {200, 300}, respectively, then the following would match: very-close, close, medium-close, 100, 200, 300, or 400.

Since the Samuel system is not a binary representation system, non-standard genetic operators are needed to process the rules. These include a *specialize* operator that tweek the values of a successful rule so that its range is more restriced, thus specialising the rule. A *generalize* operator, that takes a partial match[16] and makes it less restrictive that includes the given state (sensor reading). The *merge* operator takes two rules with the same action (right-hand side) and combines the left-hand condition parts so that the new rule matches the conditions of both of the old rules. For example, the rules

```
if (distance is [3 .. 5]) then (turn is [right])
if (distance is [1 .. 4]) then (turn is [right])
```

merge to form

```
if (distance is [1 .. 5]) then (turn is [right])
```

The *delete* operator deletes a rule from a strategy (rule set). The *mutation* (randomly change a rule) and *creep* (shift the values of a range), as well as the binary operator *crossover* completes the operators used in this version of Samuel.

The applications Samuel have been used on are based on a simple model of a fighter aircraft. These applications were simple evasion, tracking, and dogfighting. The inputs to the system (the

[16] When no rule completely matches a given input state, Samuel finds a "close match" and uses that.

sensors) are: time (since episode start), last-turn (by the agent), bearing (direction to adversary's position), heading (relative direction of adversary's motion), speed (of adversary), and range (to adversary). The actions allowed by the agent are to change speed and change direction.

In the evade test, the predator being evaded comes in from a random direction and with greater speed. With greater energy, though, the predator cannot turn as well as the agent, giving the agent a chance. The test (episode) ends when either the prey is caught or the predator loses some amount of energy, loses the advantage, and gives up. Noise is introduced into the sensors to make the learning task of the agent more challenging. The agent improved from 31% (a random walk) to 82% success after about 50 engagements.

The tracking test sets the adversary moving at random speeds and directions with the agent trying to follow. If the tracker enters within a certain radius of the prey, the adversary turns and, with a probability based on range, captures the agent. In this task, an initial set of fair rules were given the agent, since it was seen that not enough evolutionary pressure existed on the agent to evolve a useful set of rules for this task starting from a random set of rules. The initial rule set was 20% effective, and the agent improved to about 72% in 50 generations.

The dogfighting test pitted the agent against a rule-based adversary. Speed of the agent in this case was controlled by the turn being made, not directly by the agent. A combatant is considered destroyed if the heading is straight on and the weapon is within range. The agent gets full payment if it wins, some if it's a draw, and none if it is destroyed. The results showed an improvement from the random walk of 40% to about 83% after 50 generations.

Note that the above are means of 10 tests taken on the best individual. Also note that these are off-line results, since tests were on the best strategy once the learning was done.

A few other notes are relevant. The Samuel system showed a window of useable complexity, i.e. once the situation became too complex, the system failed badly. On the other hand, many

features of the system are useful to the subject of the mission-based autonomous agent that is the focus of this investigation, including condition encoding and the Lamarckian concepts.

In a later study by Ramsey and Grefenstette in 1993(56), the Samuel system becomes part of a larger system based on a technique termed *anytime learning*. The task in this system is to control a "cat" tracking a "mouse" in a simulated environment. The cat must track the mouse within the range of its sensor, but outside the smaller range of the mouse detecting the cat. In this application, the agent (cat) can control both speed and direction. An episode lasts 20 time steps, and if the cat keeps the mouse in range at least 75% of the time, then the critic (evaluation function) of the learning system gives full payment (reward). Else a partial reward is received proportional to the time the target was successfully tracked.

In this system the *monitor* looks at the 50 most recent input samples and decides if a change in tactics is needed. The learning system does this by running its internal simulation to derive candidate rule sets and evaluate them. The best 20% of these strategies are then further tested and the best one chosen as representative of the learning system. Four parameters of the execution system are monitored: the distribution of speed and of turn values (in degrees) of the target, the radius at which the target (mouse) detects the tracker (agent), and the size of the target.

When the learning system outperforms the execution system on a task, it is assumed that the internal rule base is more appropriate to the task facing the execution system and that rule set becomes the new rule set of the execution system. When the measured parameters of the execution system deviate from those of the simulation, then another rule set population is loaded into the genetic algorithm (using the percentages given next) and the GA is restarted. The rule sets chosen are in a rule set store indexed by the monitored parameters. The assumption is that a match of average speed and turn values, for instance, indicates a similar situation. The system is reloaded with rule sets in the following proportions: 50% of these rule sets are best cases stored away previously, 25% are from the previous population, 12.5% are default strategies that are always

loaded, and 12.5% are exploratory strategies (new rule sets). The GA is then restarted with the w rule set population. This is known as *case-based initialization*(56).

A key need of this method is to be able to characterize the environment using a set of parameters. This becomes a problem in a more complex environment, as is the case with many classifier system strategies. Another major detail of the system is the need for an internal simulation environment that the learning system can essentially "practice on" before passing a rule set to the execution system. A mismatch between this simulation and the real environment could be disastrous in a similar way that Pitt classifier systems become brittle as they leave the area of the state space they were trained in.

Nonetheless, the case-based anytime learning strategy showed significant improvement over classifier systems that must relearn a situation after leaving it for awhile, a result of their rule sets evolving to adapt to a new situation at the expense of the old rules (or rule sets in Pitt systems). As the situation switched back and forth between one of high speed and one of high-degree turns, the case-based anytime learning system came up to speed on the new situation faster than the basic anytime learning system without case-based initialization. The system presented at the 1993 International Conference on Genetic Algorithms was storing up to 30 such cases.

These two systems have much applicability in the design of the learning system that is the target of this investigation. More is said on this in Chapter 4.

CFS-C

Rick Riolo's CFS-C public domain classifier system has evolved to include many of the advances in the classifier field. A "Michigan" type system, it has many of the features found in both types. Here a look is taken at the various types of work that Riolo has done with the system over the years. Since this system serves as the starting point for a large number of systems (including the system proposed in Chapter 4), we examine its capabilities.

The work prior to 1989 has already been documented above, so it is not be repeated here.

Two applications that come with the CFS-C package are the Letter Sequence application described previously(61) and the Finite State FSW1 application(60). Both show how to set up and use the system.

A system of note is the lookahead planning application published in 1991(64). This system implements a simple look-ahead mechanism that improves its performance on the Finite State World One (FSW1) task first proposed by Grefenstette. Otherwise the system is very similar to the CFSC-1 system previously published and previously looked at here.

The idea is to have the system evolve not only rules that address the current needs of the environment, but also rules that can *predict* the future needs of the environment. For this system, called CFSC-2, Riolo limits lookahead to one time step (classifier cycle). By creating a new type of rule that represents ***hypothetical*** activity, and by adding other variations that allow for rule associations to be formed, he was able to generate a system that supported stimulus-response-stimulus predictions; i.e. given a stimulus from the environment and an action in response to that stimulus, the system could predict the state (stimulus) that would follow.

As in other classifier systems, the utility of a prediction is controlled by the fitness of the rules involved. Riolo takes the fitness measure of CFSC-1 and divides it into multiple fitness values that serve different functions. A long-term fitness, S_*, is maintained, similar to the fitness in normal classifier systems. It is updated by a temporal difference method (as in many other classifiers) so that its steady-state (*fixed point*) value tends to the average payoff the rule receives. This fitness value is updated every time the rule posts a message, except for transition rules posing hypothetical responses.

Another fitness value, S_t, is updated *every* step, whether the rule is active or not, and maintains the immediate utility of the rule in the current context. When the rule is active, it quickly gains strength (if it is useful and receiving payment); when it is inactive, this value is set to tend to the long-term average represented by S_*.

Finally, a third fitness value, S_p, is used to judge the predictive ability of the rule. It is 1 for all rules except for transition rules (the rules that predict the next state of the system), where it then is based on the rule's success at predicting future events. These three fitness measures are summarized below:

$$S_s(t+1) = d_s S_s(t) + (1-d_s)[R(t) + (g_s P_s(t))]$$
$$S_t(t+1) = max(d_s S_t(t) + (1-d_t)[R(t) + (g_t P_t(t))]),\ L_{st} S_s(t+1))$$
$$S_p(t+1) = d_p S_p(t) + (1-d_p) P_p$$

where d_s is a constant between 0 and 1, $R(t)$ is the reward at time t, g_s is a discount factor on future rewards, and $P(t)$ is the payment from other rules active before it. Note that $P(t)$ comes from the specific strength of that type, i.e. $P(t)$ for S_s comes from the S_s of other rules active previously. L is another weighting factor and P_p is 1 if the rule's prediction is correct, 0 otherwise.

The bid calculation is also changed so as to take into account the new strengths. This is given by

$$B_i(t) = S_p(t) * S_t(t) * H_i(t)/max_j(H_j(t))$$

where H_i is the support for rule i at time t and is given by $H_i(t) = g_I * sum(I_m)$, where I_m is the *intensity* of the message, which is provided by the detector interface or the posting classifier.

The system showed marked improvement over the CFSC-1 system in two simple finite-state worlds, reaching roughly 80 % accuracy on this test world once the system was trained (roughly double the accuracy of the system without lookahead enabled). However, the limiting of lookahead to only one time step prevented more complex predictions. Also, the genetic algorithm was not used with the system since, it is assumed, the actions would disrupt the complex rule wirings needed to link the various rules together. However, this approach shows much potential in suggesting ways to add a lookahead capability to other classifier implementations, and will be readdressed in Chapter 3.

Other classifier systems of note

Other classifier systems of potential application to the autonomous agent task are examined below.

Bruderer and Shevoroshkin have proposed a hierarchical classifier structure that addresses some of the problems inherent in complex learning tasks(5). Though incomplete as of the Summer of 1993, the structure encoded subtasks learned into essentially callable macros that could be treated as rule structures with complex actions at higher levels. These rule sequences are learned by the system as it learns about its domain[17]. They did not provide a clear way to promote the learning of such macro structures.

Work in using tags to control the generation of new rules on a classifier population was, I believe, presented in one of the working groups at the Fifth International Genetic Algorithms Conference (1993), but I have little additional information on this(76). But it is fair to note that many of the ideas presented later in this work (which were developed independently) are thought to possibly parallel this approach to some extent. Time did not permit a more thorough examination of this source.

3.5 Discussion and Direction

The goal of this chapter was to review the various alternatives that could be used to attack the autonomous aircraft agent problem, and to lay some ground work for the next chapters. This section reviews what has been seen and offers some reasons for the directions taken in this research.

This chapter reviewed various systems related to and potentially solutions to the autonomous aircraft agent task. Included was the Pilot Decision Phases in Clips (PDP-C) rote rule system, the Soar architecture, the Pilot Associate, the PAGODA system, and the missile site avoidance simulation MAVERICK. Then Animats (artificial animals) were reviewed and some applications noted. Finally, this was followed by a discussion of and some more relevant examples of genetics-

[17]The approach proposed in this thesis assumes that the subtasks to be learned, namely the mission tasks, are defined. This reduces the problem of deciding how to break up and remember the subtasks.

based classifier systems. Each of these systems addresses autonomous learning to some extent, but each have deficiencies that limit to varying extents their use in the building of autonomous agents in the simulated aircraft control domain.

The goal of this investigation is to show the feasibility of building an autonomous aircraft agent that can react with its environment in real time and learn concepts and related behaviors about it. This need for real-time learning quickly removes many techniques from consideration, such as Soar, and the need for adaptive learning points towards genetic algorithms and classifier systems as a good potential starting point.

Classifier systems, though, now come in many styles and forms, and choosing the form that best meets the needs of the designers is quite a task. This is the focus of Chapter 4, where we also present the framework for a new approach that seems suitable to the autonomous agent task.

IV. Building Adaptation into Classifier-Based Autonomous Aircraft Agents

This chapter analyzes how adaptation can be added to aircraft agents controlled via the classifier system architecture. Classifier theory is developed, various alternatives are considered, and the proposed approach, that of phased control, is presented and examined. The chapter ends with an analysis of the proposed phased-classifier architecture.

This chapter provides the theoretical framework for the building of a classifier system designed to control an autonomous agent. The design of the Phased Pilot Learning System (PPLS) based on this analysis is presented in the next chapter.

The reader is assumed to have some knowledge of classifier systems. Appendix B provides a short introduction to the basics of classifier systems and their operation, including credit assignment, tags, and chain formation.

4.1 Adaptation in Classifier Systems

In this section we build on the basic classifier system (as presented in Appendix B), and introduce rule generating strategies. These methods, which include covering operators and the genetic algorithm, allow the rule population of a classifier system to adapt to the conditions of its environment.

4.1.1 Rule discovery operators. Allocation of credit (such as via the bucket brigade or other mechanism) can adapt a set of rules to meet the needs of a particular environment. This form of adaptation, however, is limited since no new rules are created and so rules that perform acceptably in the environment must already exist in the rule population for the credit allocation mechanism to work successfully. In other words, for the agent to be able to adapt to any environment, we must somehow introduce new rules into the classifier population.

Rule discovery operators do just that: they look for weaknesses in the current rule structure and modify the rule population to fill those weaknesses. Such operators include *cover* operators, *coupled chain* operators, *specialization* and *generalization* operators, and the *genetic algorithm*. Each is discussed below.

Cover operators address the problem of what to do when no rule seems appropriate (fires) in a given situation. Two general types of cover operators used in practice (58, 60) are the detector and effector cover operators. These are designed to generate rules that "cover" situations when no rule matches an input detector message or when no rule is generating an action, respectively.

The *cover detector* operator is triggered when an environmental state is detected that is not covered by any rules currently in the system. When this happens, this operator uses the input state message from the detector to form a rule with a condition that will fire in this situation. The action is either chosen randomly or is derived from a similar rule that almost matched the input. This new rule is inserted into the population and handles (possibly not too well) the previously uncovered state. (More on this in a moment[1]).

The *cover effector* operator is triggered when the system doesn't generate an action (produce an effector message). The idea here is that by always producing actions, eventually actions that productively effect the environment will be discovered. Without this operator, and starting with a random set of rules, we may never discover a rule (within reasonable time constraints) that produces the desired action. This operator allows the system to explore new actions within the context of the environment.

Chaining operators include a variety of operators that use tags to link a rule that fires in one cycle to another that fires in the next cycle. Riolo and others argue that chains are needed to create complex, time-synchronous behaviors in classifier systems(58, 36). Chains operate by generating

[1]For the reader that can't wait, the fact that the situation is now covered is obviously not enough, since the generated rule could be useless. As we will see in a moment, the genetic algorithm is used to increase the fitness and utility of these rules. We are effectively just seeding the population for the GA

messages that trigger later rules to fire, hopefully at an appropriate time for such a firing. One problem with rule chains, however, is in finding them, since finding many rules that meet a specific criteria is many-fold more difficult than finding just one rule, this specific criteria being that one must trigger, and set the stage, for the next. Another problem is maintaining such chains, since the genetic algorithm tends to rate rules on their independent fitness, and not the fitness of the chain they belong in. Riolo addresses these problems, though not completely, in (58).

Although the above methods apply parts of previously developed rules to new situations, and hence perform some levels of inference, the primary inference engine in classifier systems has been the *genetic algorithm* (GA). GAs use parts of high-fitness rules (i.e. rules that have been shown useful in the past) to piece together new rules with potential worth. GAs are stochastic processes that are not guaranteed to produce successful results, such as a properly bounded A* algorithm would(54). However, they can be fast in comparison (polynomial verses exponential computation time) and are well suited to the rule discovery task[2].

In genetics-based classifiers, the rules are encoded as binary strings from the alphabet {0, 1, # }. This notation allows the rule to be divided up into smaller units or strings that can be operated on via *genetic operators* such as crossover (swapping of subsections) and mutation (random small changes within subsections). If we add a *selection* operation that chooses the higher-fitness (better performing) rules to participate in this process, then the forces of natural selection weed out the less fit rules (they get replaced with the offspring of the selected rules) and lead to an overall increase in the utility of the rule population. Riolo has shown (62) that GAs, when used with the cover operators just discussed, can lead to rule populations that can handle the task of navigating a learning system through the environment. These operators complement each other in that the cover operators provide the raw genetic material and the GA uses selective pressures to evolve this into useful rules of high fitness.

[2]See Appendix A for more on the theory behind genetic algorithms and some examples of their use.

The genetic algorithm requires a fitness measure to judge how well a rule in the population is doing. In this simple case, the credit allocation algorithm provides this directly. The last chapter mentioned the *profit sharing* epoch-based method, and this has been shown to be effective in many situations (see Section 3.4 for more). However, this investigation concentrates on the *bucket brigade algorithm* (BBA) introduced their and described in Appendix B.

4.2 Limiting Context, or Preventing Premature Convergence in Multitask Environments

Two of the major problems in using classifier systems to search a rule space is *premature convergence* onto rule sets that are far from optimal and *lack of convergence* onto any useful rules at all. The first can be caused by a population that lacks the building blocks to build better solutions for the genetic algorithm to try. The latter can result from a population that is too diverse for the algorithm used, providing the probabilistic search for suitable building block combinations with too large a domain to effectively search. Both of these situations can be addressed in terms of input filtering to the learning algorithms, which is the topic of this section.

4.2.1 Search space limitation. As was discussed in Chapter 2, the use of a filtering mechanism to control the examples presented to a learning system can sustantially improve its performance, turning a "weak" concept learning system into a "strong" learning system. This filtering process equates to limiting the domain of the search space to those examples that best illuminate to the learner the concepts to be learned. In a way, such prefiltering can be thought of as an example of learning from a teacher. Indeed, Schapire refers to such an example filter as an *oracle*(73:115), implying this relationship. The problem, of course, is that such functions are themselves exponential problems in the limiting case. So the task is to find such a filtering function that operates in polynomial time that also promotes faster and more accurate results in the probablistic learning algorithm.

One significant and relatively simple way to limit context is to filter the types and ranges of the detectors that the system receives environmental inputs with. This can be done in many ways; we enumerate some of them below.

Choose relevant inputs. This first filtering method simply states that the learning system should be given only those sensor inputs that are needed. If the types of clouds in the sky are not important to a submarine agent, then there is no need to funnel this information through the agent's input interfaces. Arguably all inputs are of potential relevance, but supplying too much information prevents the system from effectively learning the behaviors it really needs to survive in its complex environment.

Choose ranges on inputs that are as limited as possible. If a sensor's output is being monitored to detect the direction of a light a robot is to move to, then reporting the direction of the light to seven digits of accuracy may be inappropriate. If one digit of accuracy is enough, then the input can be filtered by the interface to supply just that information useful to the learning system.

Discretize the range when possible. Continuous (analog) inputs from sensors requires some form of determination as to which input value ranges are useful and which are not. This process is another level of concept learning that the learning system must deal with. Providing a discretized set of inputs, such as { left, right, ahead, behind } for the light tracking problem above, removes this level of concept formation from the system and allows resources to be applied to other, higher-level learning.

Provide only inputs relevant to the current context. Related to the first filtering technique, this one takes selection a step further by noting that not all inputs are relevant at all times. Rule-based systems (such as PDP-C -- see Chapter 2) use rules that trigger off of limited subsets of the possible facts in the environment. Otherwise all rules would have to check all sensors at all times, leading to an exponential number of checks (rules). The expert system designer builds in to the system this use of context limiting, noting that not all inputs are relevant once other inputs are

known. Thus an agent that is evading an enemy does not need to consider the traffic patterns in a busy intersection below (unless it might need to land in that very intersection). How to choose what to look at and who chooses it are important issues to address here.

4.2.2 Restrictive mating. ***Restrictive mating policies*** were proposed by Booker in his 1982 dissertation and were described in Chapter 2. In short, this method limits genetic search operations to those population members that are currently active as a result of matching a message or messages on the message list. He argues that classifiers that are active are more likely to be applicable to the current situation than other classifiers in the population, and that these active classifiers contain more useful building blocks than a random selection from the population would[3].

In Booker's systems, he makes a number of changes to the basic operation of the classifier system. First, he breaks the environment into two distinct parts: an environmental model part and an action model part. Inputs from the environment are received by the environmental part, which processes (filters) these messages and presents a derived state to the action part of the system. These perceived state messages are then used by this part of the system to determine an appropriate action for the given state. This separation of sensing from action allows a level of abstraction (i.e. classification of inputs as concepts or objects) to develop within the system.

Another modification he makes is adding a *restrictive mating* policy that restricts genetic operations to only active classifiers. To enhance the mating populations he implements a *matching score* that allows rules that *almost* match a set of messages to *partially* fire. Though this approach adds an extra level of complexity to the matching operation[4], it allows rules that are close to provide genetic material into the mating pool. This approach, however, has problems when applied

[3] See Chapter 2 for more on Booker's specific systems. The current discussion will concentrate on the techniques he uses.

[4] To count up the number of matching bits and calculate the match score.

to problems requiring chains of actions to form, since such a mating policy only looks at one link in the chain at a time, and provides no method to link these restricted populations together[5].

4.2.3 Triggered memory. Another way to limit the effects of the discovery algorithms on the rule population is to save away the active rules of an epoch[6] into some form of "long-term" storage and call them back when a similar situation is encountered. Many questions arise, especially how to define when a set of rules is "worthy" of being stored as an environmentally adapted set and the *triggers* used to determine when that set of rules is again useful.

One example of this approach is Zhou's Long-Term Memory Classifier(84). This system was designed to handle the *reversal task* where an agent in an environment finds the sources of reward and punishment suddenly reversed. Since the rule population is now laded with the wrong rules for the task, the basic classifier system now must learn a new set from scratch. Zhou's system was designed to save the old rules in a long-term store and bring them back when the situation again reversed.

Zhou used what he called *descriptors* to classify th knowledge saved in the rule store[7]. A full or partial match to these descriptors then could be used to trigger the injection of a subset of rules (learned previously) into the existing rule population. Thus, after a sufficient store of rule sets are stored and categorized, the reversal task becomes relatively simple.

Zhou did not address many issues, however, and did not implement a system using them (to my knowledge)[8]. His system assumed that tasks could be distinguished and that the classifier system only had to deal with one task at a time. In a complex environment this may not be the case. Also, he gave no guidelines on how to choose descriptors in such complex situations. However,

---

[5]Booker's work focuses on stimulus-response systems (with an added level of abstraction) so does not address this issue.

[6]The definition of "epoch" is slightly different here in that payoff might not be involved in determining its cycle boundaries. In fact, one might swap in a new set of rules whenever no progress is being made.

[7]I will use the term *rule store* to refer to the long-term memory in such systems. The reader should not confuse this with the short-term store represented by the changing rule population of the standard classifier system.

[8]Interestingly enough, Zhou was being advised by Grefenstette during this period, who was involved in a similar system presented next.

many of the concepts in this paper are still worth applying to the autonomous agent problem. More on this later.

Davis, Wilson, and Orvosh use a similar scheme in a new version of the BOOLE system, a one-step classifier[9] that classifies boolean functions given a set of inputs and resultant outputs(11). In addressing the *multiplexer problem*[10], they used a classifier system with an added memory that stored "examples", a list of triples of the form (weight stimulus response), where *weight* is a weighing factor of the tripple's usefulness and the other two values the input and output of the example. Memory was limited to a fixed number of examples. The system performed the following steps:

1. Read an example from the environment.
2. For some number of cycles, select probabilistically by weight examples from storage. Present these to the classifier and then run the genetic algorithm to modify the classifier rule population. Decrement the weight of the rule store example.
3. Repeat.

So rule examples are tried a number of times based on their relative weights, each example training the classifier system to some extent.

This system showed improved results on the function classification task, but uses a single task system similar to Zhou's and is only partially applicable to autonomous agents in a changing environment. It also had the advantage of avoiding the descriptor issue by just trying all examples in proportion to past performance, but this approach is of marginal use if external trials are limited and an internal simulation (world model) is not available to try the examples on.

Grefenstette's Samuel system discussed in 3.4.2 (page 3-22) looked at a similar approach to allow a simulated cat to track a simulated mouse in differing situations(56). This method uses a

[9] A *one-step classifier* provides output on the same step as presented with the triggering input and uses no chaining.

[10] A standard problem in concept classification circles where some of the inputs to a function select which bit of the remaining inputs is passed to the output. Since it is not based on a specific easily-derived mathematical relationship, it proves to be a good test problem for classifier systems(11).

monitor to select when to load in a set of rules based on various environmental parameters that were assumed to describe the current state of the environment (such as the speed of the mouse and how often it turned). The system depended on this set of characteristics to be unique enough to identify the situation. In many environments this is not possible, especially those where the significance of the various detector inputs varies over time (as different tasks are to be accomplished, for instance).

4.2.4 Hierarchy-building systems. Some classifier systems have tried to address the complexity issue by building hierarchical levels into the systems. These systems chunk information into essentially macro form where it can be recalled as needed (i.e. when its trigger conditions are matched) and used at a later time

4.2.5 Phasing - another approach to restrictive mating. Another method[11] hinted at in the literature and discussed on occasion at conferences(76) is the use of tags to restrict mating. The idea is simple enough. Each rule in the population is assigned a set of tags that represents its *phase* of activity[12]. The phase tag acts as a normal tag, preventing the rule from matching messages outside the context of the rule's phase. Thus each phase forms a cluster or family of rules that are active only during their phase.

Each condition, action, and message field in this approach have the same encoding, allowing action messages to be fed back to the message list as in standard classifiers. Besides the type field that determines the type of a message (from detector, action generated, effector destined), two new fields are added: phase and new phase. The phase field j notes the phase context of this message. The new phase field is discussed below. The type and phase fields are typically under control of the detector interface in that these fields are provided by the interface.

The detector interface is modified to encode the phase automatically into the messages being received. This provides a locality mechanism controlled by the detector interface. The current phase

[11] The material presented here is primarily that of the author and so has no specific references.

[12] This tag should not be confused with the chaining tags discussed in Appendix B. Those tags can be considered, and are implemented here, as separate fields within the encoded rule.

is changed by the system using a special action operation called PHASE. The operator operates like the standard pass-through operator, except that the new phase field of the action part of the rule is used to determine the new phase. This allows the rule population to control the current phase of the system and switch context (rule populations) on cue from external (detector) events. The potential for triggering phase rules off of internally generated messages is also possible.

The system is further modified to limit rule selection during discovery operations to only those rules within the current phase. The assumption here is that rules within a phase form a *niche* that addresses a specific task of the rule processing system. Note that this niching mechanism is separate and theoretically compatible with the active message restricted mating policies of Booker(4). Thus it is possible to develop subniches inside the phase niches. The two functions modified are parent selection and selection for rule deletion. Otherwise the discovery algorithms work as in the standard system.

This approach is useful in situations where a set of predefined tasks must be handled. Each task is given a phase and the system then jumps between phases as PHASE operator rules detect mission changes. The approach is similar to the phasing in PDP-C(31), but the phase transitions and the contents of each phase's niche are modifiable by the discovery algorithms.

A further enhancement is the creation of a REWARD operator to allow the rules to essentially pay themselves when a payoff condition is detected (such as the target being destroyed). This adds a level of complexity to the discovery process and is not considered further here.

Finally, by adding a RETURN operation, the system can essentially act as a standard program. For example, the learning system might detect an approaching enemy aircraft, switch to a defensive phase, handle the situation, and then a RETURN rule that detects situation resolution would fire and return the phase to the previous one. If a stack mechanism is used to store the previous phases, then essentially infinite levels of interrupt could be supported. However, as the mechanism becomes more complex, the ability of the system to learn appropriate use of it is re-

duced, due to the increase in domain complexity. This would have to be considered in implementing this approach in real classifier systems controlling autonomous aircraft agents.

4.2.6 The potential of self-sensory-restricted systems. In Chapter 2 it was noted that animats have been designed that limit the sensory inputs to the system to only those needed to do the task it is currently doing(48). This reduction of inputs reduces the search space the system must deal with, and so should make the learning of useful behaviors in such systems easier. Can this be applied to classifier systems?

One implementation of this approach could use the above phasing mechanism to select the detectors encoded into the detector messages. Since each phase is isolated from the others, this would allow the overall complexity of each phase to be reduced, in effect providing a strong method of context filtering.

4.3 Interfacing to a DIS environment

This section departs from the previous sections and focuses on the interfaces needed to connect a learning system to a simulation network.

We apply the ideas presented here in the next chapter, where they are used to design the prototype learning system's interface, and again in Chapter 7, where we examine them in a broader context.

Distributive Interactive Simulation (DIS) is a proposed standard for interconnecting distributed simulations within the bounds of a synthetic world via network communications links(19). More specifically(16:1),

> DIS is a time and space coherent synthetic representation of world environments designed for linking the interactive, free play activities of people in operational exercises. The synthetic environment is created through real-time exchange of data units between distributed, computationally autonomous simulation applications in the form of simulations, simulators, and instrumented equipment interconnected through standard computer communicative services. The computational simulation entities may be present in one location or may be distributed geographically.

The applications[13] that DIS support are many, and include manned vehicle simulators, computer-generated forces, and computer interfaces to real equipment. By exchanging packets of information called Protocol Data Units (PDUs) between these entities along the various network media, the goal of DIS is to create a complete synthetic battle environment that users interact with in real time (19:1).

DIS is a distributed protocol with no central control. This means that exercises between applications (that support one or more agents) can be spread between various hardware systems. A ground truth model is used, meaning that actual location and other data is communicated via the networks and the receiving applications given the task of presenting the perceived version (if any) of the simulation object to the simulation agents it supports. *Dead reckoning* techniques are used by the receiving applications to maintain the current position and status of such remote simulation objects between packet receptions. Also specified, among other things, are the world geometry, weapons fire, and communications protocol(19:2-7)

One of the goals of this investigation is to present a set of guidelines for the interfacing of an autonomous agent to a distributed simulation. We argue that any such interface must be distributed to be scalable. If we define a ***distributed interface*** to be one where no part of the interface structure performs more than a relatively simple, coherent task, then each such part can be implemented as a pseudo-process within a possibly distributed computer architecture. Before we continue with this development, however, we digress and examine the various interpretations that can be used to view such a structure.

4.3.1 Pipes and filters, layers and servers. Figure 4.1 shows a general layered interfacing structure. In this interface arrangement, information is received by the network interface, processed by different layers, and finally delivered to the agent as its *stimulus* or environmental inputs. The

[13]This definition is slightly different than we have used so far, i.e. an agent within an environment, but this difference is mainly a change in perspective.

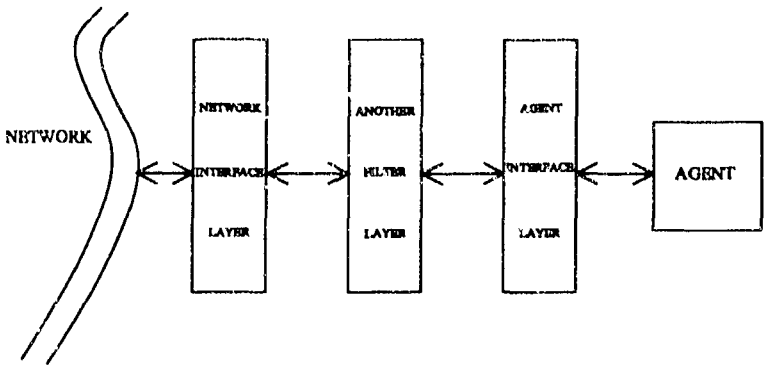


Figure 4.1 Connecting an agent to the world.

agent then reacts in some way to these inputs and generates a set of outputs or *response*, which is then passed back to the network interface and broadcast to other agents in other simulations.

As inferred above, one way to look at this problem is as a set of layers, each processing the data and passing it on to the next layer. Each layer can be thought of as encapsulation of the functions of that layer, providing a transaction-level conversion to data flowing through the layer. Entities on a particular side of the interface only see a specific model of the other side presented to them. In such a design, the entity only has to meet the protocols of the interface and does not have to worry about what lies beyond it. This is the view taken when DIS defines the *application layer* as one layer in the DIS structure. It is also the approach used in the Open Systems Interconnection model(78:86,100-105).

Another way to view the interfacing problem is to view each interface between data representations as object managers with inputs and outputs, similar to how an Object-Oriented Design defines the entities in a program to be objects that receive information, react to it, and generate a response that it passes on to other objects(7). This view emphasizes the desirability for loose coupling between the various entities to reduce traffic flow and keep the interfaces simple.

Yet another way to view the flow of data is to treat each interface as a *data filter* that filters and converts the input data into a form usable by the next receiving stage. This is exactly equivalent to how a pipe (in the Unix context, for instance) is composed of a set of filters that each receive

input, process it, and pass on a new version of the data to the next filter in the pipe. In this view, a set of filters converts the data from the DIS network into a (typically) simplified form that the agent's learning system can use. The agent's learning system makes a decision about this simplified form of the data, then passes this decision back through another set of filters to the DIS interface. By the time the data reaches the interface, it has passed through a flight model, for instance, that has generated the "real-world" consequences of the agent's decision, and a PDU generator that has converted the data into packets for sending over the network.

Finally, some parts of the system can be viewed as *servers*, in the parallel computing sense of "an object plus a task"(45:155), that receive inputs from many sources and can provide services to many other simulation objects based on these requests for service. If we view the server as "an object in execution"(45:155), then servers provide a cross between a layer and a filter that react to structured data presented to them and pass on structures that encode the results of these reactions. This view lends itself to parallel architecture distribution if the servers are loosely coupled.

The reader should note, however, that each of these representations are of the same system, and that each, if generally enough specified, can be made equivalent to the others. The idea is to view the problem at hand using the model or models that makes the problem the simplest.

4.3.2 A suggested interface structure. We now present a possible interfacing structure for connecting the autonomous agent to the DIS network[14]. The philosophy taken is to minimize the amount of processing at any interface in the system. The reason for this is to facilitate the debugging and later maintenance of the parts of the system. Each interface layer can be tested individually, as can the pieces in a good software design, and then assembled and tested as a whole. And, as with object-oriented designs, modifications to the system can be isolated to the

[14]This proposed design is mainly based on discussions with Daniel Gisselquist and Dean Hipwell during the design of the PPLS system. Dan Gisselquist was especially helpful in designing this interface.

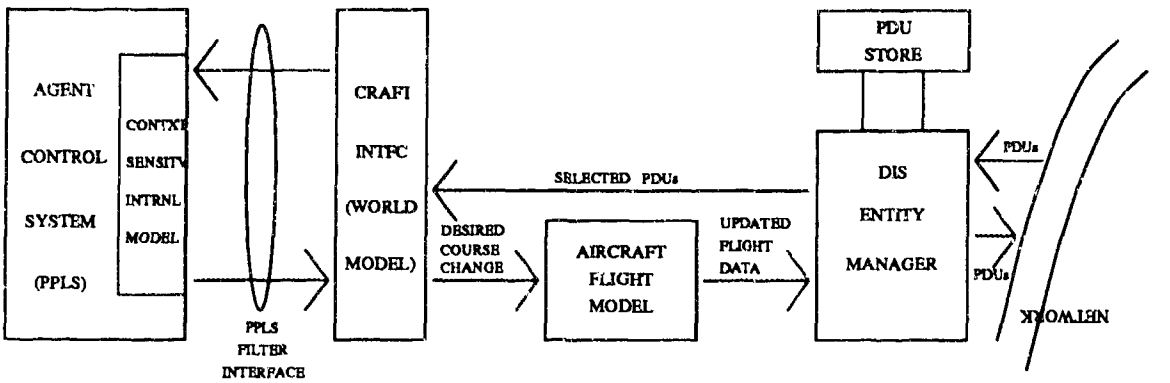


Figure 4.2 A proposed DIS interface.

various interface layers. This facilitates the adding of new interfaces to other simulation systems, for instance.

PDUs are buffered in the *entity manager*, which maintains a store of the various objects seen on the DIS network. This data is filtered according to the requirements of what I call the *craft interface*. Only those simulation objects of concern to the autonomous agent controller (the learning system) need be passed to the craft interface. The entity manager therefore filters out any PDUs, (such as ground activity) that the autonomous aircraft agent doesn't currently need.

The craft interface acts as another filter, converting the complex state data to a simplified form, taking into account the current needs of the learning system. Any data not needed by the learning system to make the current decision need not be presented to it. This is an important point. We distinguish here between the cookbook filtering of the craft interface that can take a deterministic amount of time and the heuristic searching of the learning system. Even though the craft interface may have a lot to do, the possibly exponential time learning system can still be the bottleneck in the system. If the context of the system is under control of the learning system (as proposed in the PPLS system of Chapter 5), then the craft interface and the learning system work together to learn filtered concepts in polynomial time. This assumes the craft interface provides the sufficient filtering noted by Schapire to adequately isolate the concepts to be learned(73).

Likewise, the simple decisions of the learning system are filtered through an aircraft model[15] and the resultant flight behavior passed back to the entity manager. This, in turn, passes the new state to the network, and also back to the craft interface, if needed, to provide feedback on the current state of the self craft. By using filters at all stages, each interface is reduced in complexity, loosely coupled, and easy to test.

4.4 Summary and Discussion

This chapter has presented much of the theory behind rule discovery in standard classifier systems and many enhancements that may be useful in our system. Important are the ways that context is limited in classifiers to allow the learning system to focus on a filtered domain space. The Samuel system (Section 3.4.2) provides a good example of the need for context limiting in learning systems.

One question to ask is if rule chaining is really necessary to produce complex behaviors in classifier systems. We argue that the answer may be no, especially if the inputs to the system provide enough previous state information. This is equivalent to a human reacting to the perceived immediate state of the world based on the states of all things in the environment. Only occasionally, we argue, does a human actually think through a set of steps to arrive at an action. Most actions are instantaneous based on the immediate situation. Likewise, if the learning system is given a set of suggested actions and a sufficient environmental state model, the decision can be made on these inputs alone without reliance on a chain of reasonings. Concept learning is just this process of deciding if what is seen is indeed what we want.

[15]The model presented in *NPSNET: Flight Simulation Dynamic Modeling Using Quaternions*(8) by Cooke, *et al.*, currently provides the flight model for some of the latest AFIT simulation designs. It also presents a way to determine the settings of flight surfaces to have the simulated aircraft, say roll right and climb, by solving a small matrix.

Population niching allows the classifier system to focus on the specific needs of the current situation. Booker's restricted mating policies allow such niches to form within a population. These methods show promise and are considered in the design of the PPLC system.

The phasing approach developed by the author provides another means to limit context in a multi-mission environment. As long as the types of missions can be defined, which is generally the case in the autonomous agent problem, this approach can provide the learning system with a simplified environment representation that lends itself to concept formation. It also allows the system to jump between phases as the situation dictates, even providing a form of "interrupt" capability that can handle when a high priority situation is detected and must be immediately addressed. This approach is similar to hierarchical classifier system designs, but implements the hierarchy at a different level.

The design proposed in the next chapter therefore uses a phasing system to form subpopulations that handle each of the tasks of the mission. Although the phases themselves are hard-wired into the learning system at present, when each population niche is called upon by the system is completely determined by the rules in the system and the inputs form the detector interfaces. This should allow for very flexible behavior (and an unfortunate increase in search space complexity).

The remaining chapters address how a prototype system using these techniques system is implemented, how it is tested, and the results of this testing.

V. *Design and Analysis of the Phased Pilot Learning System*

This chapter details the Phased Pilot Learning System (PPLS), including what the system does and how it is interfaced via a layered hierarchy of interfaces to the target environments. We also present a formal software analysis of the implemented system.

The design of the PPLS system must take into account the different aspects of learning. For instance, new rules should be implantable into the system as information is available. Alternatively, in situations where new rules are not known, the system should be able to explore its environment and learn such rules autonomously. In any event, the input and output interfaces need to reflect as clearly as possible those parts of the world important to the PPLS learning system. The better this filtering, via methods of deduction and data reformulation or even other learning systems, the easier it is for the PPLS learning system to adapt its rules to the environment. The formalities of problem analysis and program design are addressed in this chapter.

Important to the design is the need for an effective user interface. PPLS is based on an enhanced version of the CFSC-1 public domain classifier system of Rick Riolo(59). The modifications give the system a means to filter out rules that are not needed during a given phase of a mission, enhancing the learning system's ability to discover and remember rules that prove useful for each phase of the mission. The specifics of phasing in PPLS are presented here, along with the rest of the design.

The environments that PPLS are interfaced to include a very simple test environment that maintains the self craft only (all other objects don't move). A second interface to the PDP-C system is provided(31, 32), giving the system access to and control of an independent agent in the PDP-C simulation. This is a complex interface with some unique synchronization requirements. A third interface to a monitoring program allows agent activity to be monitored graphically each iteration, providing the user with immediate feedback on all activity. Finally, a fourth interface discussed, but not as yet implemented, is to a proposed DIS network "entity manager" currently being designed

and slated for completion in October 1994. Each interface provides a unique challenge and a unique perspective on what PPLS can do.

5.1 Program Analysis

This section looks at the problem from the program's point of view of data structures and data representation. A set of requirements is presented. Then the data structures are determined.

Problem Requirements

The specific problem being addressed by this research is to determine the feasibility of using a standard classifier system as the control system in an adaptive autonomous aircraft agent. Specific criteria have already been noted in previous chapters, so what is needed are the precise functions the system is to perform. These are

- The system is to store and process production rules that can control the behavior of a simulated autonomous aircraft agent. The use of production rules is dictated by the use of a standard classifier system as learning system. The encoding of the rules can be either symbolic or binary, though basing the system on the CFSC-1 classifier package strongly supports binary encoding since this is all the base system supports.
- The system is to show the ability to learn simple tasks via its discovery algorithms. These tasks include the target bombing problem described earlier (and in detail in the next chapter) and enemy evasion.
- The system is to provide a structured and flexible interface to the environment capable of easy reinterfacing to other environments. The structure of this interface should permit easy system expansion and distributed processing as much as possible. To this end, an object-oriented approach to the interfacing design is a good potential methodology for this.

- The system is to provide a reasonably simple and useful ("user friendly") interface to the user. This requires quite a few enhancements to the CFSC-1 system.
- The system must allow for adequate control and monitoring of the system to gather results showing its ability to accomplish these tasks. Additional monitoring tools are to be developed as required.

Program Structures

Given the CFSC-1 classifier system (described in Sections 3.4 and 4.1, and Appendix B) as a starting point, the data structures for the internal workings of the system are more or less determined. For completeness, these structures are summarised here.

The basic two structures in the system are the message list and the classifier list. Both lists are implemented as linked node structures. The actual condition and action fields are implemented as encoded unsigned integers to allow for quick determination of match conditions. Since the structures can be any number of bits in length, a system that encodes them into as many words as necessary allows for quick matching in a flexible way. The trinary alphabet { 0, 1, # } is represented by two sets of unsigned words. The 0, 1 status is noted in the first set of words, while the "don't care" status represented by the # is noted in the second bit array as a mask for the compares. Routines are provided to convert the structures to an ascii format to allow for easier processing of action operations, display of values, etc.

The basic functions are as noted in the above referenced sections.

Interfacing Structures

These are addressed later in this chapter, and so are not discussed here, except to say that the interfaces are a simplified version of the proposed interface design given in Section 4.3.

The program interfaces to both a test simulation and the PDP-C simulation system. These interfaces are presented shortly. The DIS interface design is similar to that presented in Chapter 4.

5.2 Software Complexity

This section examines the complexity of the PPLS system, which includes an analysis of parts of the CFS-C classifier package that the system is built on. This analysis then looks at the potential to scale the system, as implemented, up to more complex tasks.

The PPLS system is composed of the following files with the following complexities. A hierarchical arrangement is used to show what files call what other files. The columns at the right include function complexity and total complexity (including all called functions) at that level.

Note that this analysis is only for the Classify loop, the main loop that executes when the core classifier system is running. Other functions involving system input/output, etc. are not included since they do not affect the running system's operation.

The symbols are defined as:

St	Number of Classify steps executed
Nc	Number of classifiers in population
Nm	Number of messages in message list on a classify cycle
Sz	Number of bits in a condition or message
Sw	Word size in bits
Sb	Number of words in one condition (packed representation)
Nd	Number of detector messages
Ne	Number of effectors implemented
Nop	Number of Classifier operations defined

The complexities for the basic classifier loop are given in Figure 5.1 and for the discovery learning algorithms in Figure 5.2.

If we assume that Nm < Nc on any given cycle, then the complexity of executing St classifier steps is St * Nc * Nm^2 * Sz^2. The discovery algorithms (in Discover) adds to this when they

Function			Complexity (This module)	Complexity (Inclusive)
Classify			St	St * (Nc + Nm) * Nm^2 * Ss^2
	StartStp		Nc + Nm	Nc + Nm
	RdDetect		Nd	Nd * ?
		GetDMsgs	variable	variable
	GenCands		Nc * Nm * Nm * GenCand[Note 1]	Nc^2 * Nm^2
		GenCand	1	
	GenNMsgs		(Nc + Nm) * PostCfMs	(Nc + Nm) * Nm^2 * Ss^2
		CalcBid	40	
		PostCfMs	Nm * Ss * Nm * Ss[Note 2]	
		MakeNMsg	1	
	GenBehav		Ne * Nm * Ss	Ne * Nm * Ss
	UpdCfStr		Nc * 30	Nc^2 * 30 * Nm
		ApplyBB	Nm	Nm * Nc
		PaySuppl	Nc	Nc

Figure 5.1 Complexities. Complexities of the classifier loop routines.

Function		Complexity (Total)
Discover		Nc^2[Note 3]
	DscCDM	Ss * Nc
	DscCEff	Ne * Nc
	DscACPC	Ss
	DscCSS	Ss
	DscTLB	Cf
	DscTLB1	Cf
	DscBkgGA	Nc + Nc * Ss
	RplcCfs	Nc^2

Notes:

1. One of the Nm factors goes away if only one condition field is used (WildCond2 = 1).
2. If OneMPerC = 1, then only Sw * Sb + Nm.
3. The actual complexity is dependent on which discovery algorithms are selected. Also, these algorithms are not typically executed every cycle.

Figure 5.2 Complexities (cont.). Complexities of the discovery learning algorithms.

execute, but only execute on occasion. The overall complexity is based on the number of rules in the system times the square of the allowed number of messages. In a typical system, the number of rules is much larger than the number of messages, so these two quantities are roughly equal. The complexity is then roughly the cube of the number of rules. (Note that most operations are performed independently on each rule. This allows the system to be parallelized relatively easily by distributing the rules over many processor nodes.)

Since the interfaces to the outside environment only map one form of parameter to another, for the most part, the main concern of these interfaces is communications time. This is quite varied on the Sun network, but should be much more stable and predictable on a separate network, as is proposed for running DIS. Also, the use of interprocess pipes removes the file system delays, if all processes can be synchronized and all can be run on the same hardware architecture. Note that log files generally prevent running with no file system accesses.

5.3 Implementation

This section details how the PPLS system is implemented. The details of the CFSC-1 system components was given earlier in Section 5.1. Various interfacing structures needed by the system to communicate with its environments are detailed in a later section.

5.3.1 The CFSC-1 System. The Classifier System in C (CFSC-1) classifier package written by Rick Riolo of the University of Michigan was used as the baseline for PPLS. The modifications required are documented in the source code provided as Appendix ??. This package of software routines implements many of the algorithms found in the literature and provided a good starting point for implementing this system.

The package has undergone many iterative changes since it was first developed around 1986(59). As a public domain package, many of the latest techniques have been incorporated into the system as they were developed by various authors. These include some of the mechanisms used

by Booker's restricted mating policy approach, a crowding mechanism based on DeJong's work, and the chaining and covering operators developed by Riolo himself. These are all controllable via a set of system variables under user control at runtime.

The flip side of this is that the many revisions have left parts of the source code rather hard to decipher. Though mostly well commented, the core routines have grown and now interface to many parts of the system. This impeded efforts to implement the changes below, since many various side effects sometimes resulted from the simplest of changes. For instance, limiting the selection of rules as parents (as was done to implement part of the phasing mechanism), resulted in no rules matching in some cases. This made the system unusable until the cause was determined and fixed.

Overall, however, the system provides a good starting point for Michigan type classifier implementations. See the source code in the Appendix for further details on specific implementation details.

5.3.2 System modifications. The modifications can be summarized as follows. Most modifications can be disabled by changing the status of one of the modified system's variables.

- The system was converted to allow for single-classifier operation. This allowed the routines to concentrate on stimulus-reaction responses and minimize the effects of internal messages. This, though, does not mean that the approach is not usable with classifiers with multiple condition fields, since the WildCond2 variable controls this operation.

  This change was implemented to limit the system to a simple, non-chaining configuration. The second message field in the CFSC-1 system is generally used to match against messages generated by other classifiers the previous cycle step. Also, by removing the second level of matching from the core loop, the system complexity is reduced by Nm.

- An internal environment was implemented to conduct tests on the ability of the system to learn. This is an extremely simplified environment, but complex enough to force specific actions to be learned.

  The test environment uses the same file interfaces as the PDP-C interface, but uses an additional routine inside the interface to simulate the movements of the self craft. The activities of the enemy Bogey are limited (with location generally fixed) to minimise the complexity of the system during cycel step analysis. The details of this environment are given in a later section.

- An interface to the external PDP-C simulation was provided, allowing the learning system to directly monitor the actions of and to control a single aircraft in the PDP-C simulation (described in the next chapter). This is a full-scale simulation that implements multiple rule-driven aircraft within a DIS-compatible environment. The interface uses Unix pipes and semaphore files to synchronise activities with the PDP-C system. This environment is also detailed later.

- The phase mechanism was implemented. Routines affected included the matching routines, the discovery algorithms, and the environmental interface. This mechanism was implemented in such a way that the other mechanisms built into the CFS-C system still can be used, allowing multiple levels of learning.

- Various other modifications to support the application and the provide a smoother interface (including an alias processing system, a wildcard variable search system (for the 200+ variables the user can set), and the moving of the environmental variables to the main list to ease their access)).

Single classifier operation. The conversion to single classifier operation allowed the system to operate on rules of the form cccc/aaaa, instead of the two condition field form. This facilitated the stimulus-response approach being implemented by this investigation and also reduced the com-

plexity of the main classifier loop. It also helped to reduce the display requirements of the system, since even Sun workstations have problems displaying rules with two 32-bit condition fields (broken down by field) and a 32-bit action field (also broken down).

Internal environment. The internal environment simulated the environment of an outside world by reading in the output file of the PPLS system, processing the simulation objects in the file, and saving the file to the input file interface of PPLS. This approach allowed the system to mimic the operations of the PDP-C interface, for instance, and permitted the testing of many of the routines shared between the two interfaces.

Key to this approach was the implementation of a *craft structure* layer that maintained the state of the outside world in an environment-dependent structure. The PPLS system then used two routines to map this structure to the internal PPLS data structures and convert a modified version of this internal structure back to the craft structure format. This approach allowed the PPLS core system to adapt to various interfacing requirements without any drastic change to the core routines. All conversions and special interfacing requirements were handled at the interface layer that maintained the craft structure.

To facilitate monitoring the test system's activity, actual files were used to implement the environmental interface. This approach slowed the system down greatly, due to the conditions of the AFIT file server structure, but allowed a thorough analysis of the system's activities during each time step.

External interface. The external interface mainly focused on interfacing with the PDP-C system. This system (described in more detail in the appendix) provides an external file-driven interface by which an outside system can control one of the simulation objects in the simulated world it maintains. The system is written in the rule-based Clips/COOL language, and the interfaces reflected this by passing to the PPLS system a COOL object structure that represented the agent's object in the simulation. Control was provided by allowing the PPLS system to make changes in

this structure (such a increasing throttle or changing angular accelerations), and then letting the PPLS system pass the object structure back to the PDP-C simulation. The simulation system then made the modifications to the agent's simulation object and executed a simulation cycle.

Synchronization was provided using a set of Unix pipes – one for input and one for output. A set of semaphores were also established inform each participant in this protocol when information was available. This approach allowed the two system to run lock-stepped, which facilitated analysis of both systems' behaviors.

Phasing mechanism. The phasing mechanism was implemented by creating two new fields in the condition/action field structure and by modifying the operation of some of the core CFSC-1 routines. The two added fields, *CurrentPhase* and *NewPhase*, provided context tagging of both messages and rules. The detector interface was modified to add the phase tag to messages as they were built from environmental inputs. These tagged messages prevented messages outside of the context of the current phase from becoming active during any particular phase. Since only rules whose phase tags matched the phase tags of input detector messages can fire, this effectively split the rule base into a set of subpopulations that represented each phase of mission activity.

The core CFSC-1 system required some modification to implement this system. A new actior operation, PHASE, was created, that allowed a rule to change the current phase of the system to a new phase given in the *NewPhase* field. By triggering these phase change rules when specific mission criteria were met, this approach implemented a sort of agenda that specifies the different mission phases the agent has available. Since these rules are conditionally triggered, modifications to the agenda can evolve as the mission executes. This also implements a hierarchical rule structure, since each subpopulation of rules specializes to a specific part of the mission This allows a sort of program structure to be implanted also, or even to evolve.

The discovery algorithms were modified to implement a form of restrictive mating based on Booker's approach, but limiting the discovery operations to those rules within the current phase.

As in Booker's methods, the assumption here is that rules that are currently servicing a particular mission requirement (environmental niche) are more likely to contain relevant building blocks than other rules in the population. The parent selection and the rule replacement algorithms were both modified to enforce phase mating.

Other modifications. A number of other modifications were made to the CFSC-1 system to allow monitoring and control of system operation. These are described below.

The system maintains a large data base of runtime variables (over 200). Since many of these variables are cryptic in both name and function (such as bkggart for Background GA execution Rate), an aliasing feature partially implemented in CFSC-1 was expanded and a description field added. Most variables were given more descriptive names (once their function was determined) as well as a description as to what settings were appropriate for them. The Display function was modified to allow for wildcard matching, allowing the user to determine which variables were applicable in setting up the cover operators, for example. The display interface was also modified to show the alias and description fields as well as the variable name and value.

A plot file utility was added to allow the progress of the agent through the environment to be saves and later plotted using a utility such as GnuPlot. The track of the self agent was saved, as well as that of the enemy, and the locations of target and base destinations and ordnance drops included. This proved to be a useful analysis tool.

An interface to a real-time monitoring utility, showpdpc, was added so that the progress of the agent in the simulation could be examined as the simulation executed. This proved valuable when the time came to match the rules that were firing with the detector messages and the current state of the agent. As different phases executed, the agent's track showed the variations of mission focus. For example, when the agent noticed an enemy in its path, its track quickly changed to avoid the Bogey. Once outside of a specific range, the previous phase's rules took over and the track adjusted to steer the agent to the current destination.

A symbolic rule interface was added to allow rules and messages to be viewed in a more readable form. The input rule routines were also modified to allow either binary or symbolic rules to be read in via the rule loading routines. This interface uses a lookup table to translate symbolic names to binary loci in the rule fields, and so is completely redefinable for use with other applications.

The environment-dependent code was moved to a separate set of files to promote the distributed interface concept. One exception was the addition of the environmental variable structures to the global variable list. This blurred slightly the interface (though only in two localized routines), but allowed the powerful wildcard display mechanism to be used to display, set, and save and load all environmental variables.

A new action operator, STOP, was added so that execution of the system could be terminated based on the triggering of a rule's conditions. This allowed a test to be performed until either the termination condition was met or the maximum number of cycles were executed.

Other changes. Finally, there were many parts of the code that were just in error and needed fixing. These included uninitialized pointers, rate determination conditions, and many other minor and not so minor bugs. Fixing some of them required an indepth look at the function of various sections of code. This is one of the costs of using a system the author himself says is in a constant state of development.

5.3.3 System startup and execution. The environment is set up via a set of files read in after startup. The first, init.cfs, establishes the names and descriptions of all run-time variables and contains the names of the startup classifier, message, and environment files. This sets up and initializes the system.

Once the system is started, a previously saved rule set can be loaded to restore the rules and fitnesses to a state previously saved by the user. Also, command files can be read in to set up

various autodisplay functions and set various environmental and logging variables. The plot files are also initialised at this point, if selected.

The system then waits for the user to execute classifier cycles via the `classify` command. Any number of steps can be implemented, though single stepping is useful when monitoring the exact interchange between the PPLS system and the environment. When the PDP-C environment is used, the system freezes (due to the pipe operation and semaphores) until the PDP-C system responds. This allowed the checking of both systems in a single-step fashion.

The monitor facility is turned on via a set of system variables. Activating this interface instructs the PPLS system to generate a separate data file that can be monitored as the system executes. (Remember that if pipes are used for the main communications links, this inhibits direct monitoring of the input and output interfaces.) A system of semaphores can be turned on to ensure data is completely written to the interface before the monitoring utility reads it.

At any point the user can either save classifiers or messages to a file, or reload them from a previously saved file. The environment is harder to save, since many aspects are dependent on the simulation on the other side of the interface. The exception is the internal test interface, whose state is completely determined by system variables that are saved, the rule and message populations, and the interface files.

The system terminates when the user enters the STOP or QUIT commands. The log and plot files, if still open, are also closed at this point.

5.4 Object-Oriented Design Aspects of the Phased Pilot Learning System

This section looks at how the design of the PPLS system follows many of the constructs of object-oriented design (OOD), though most of the CFS-C system itself uses a hierarchical functional breakdown. We also look at what would be required to reimplement the system from the OOD perspective.

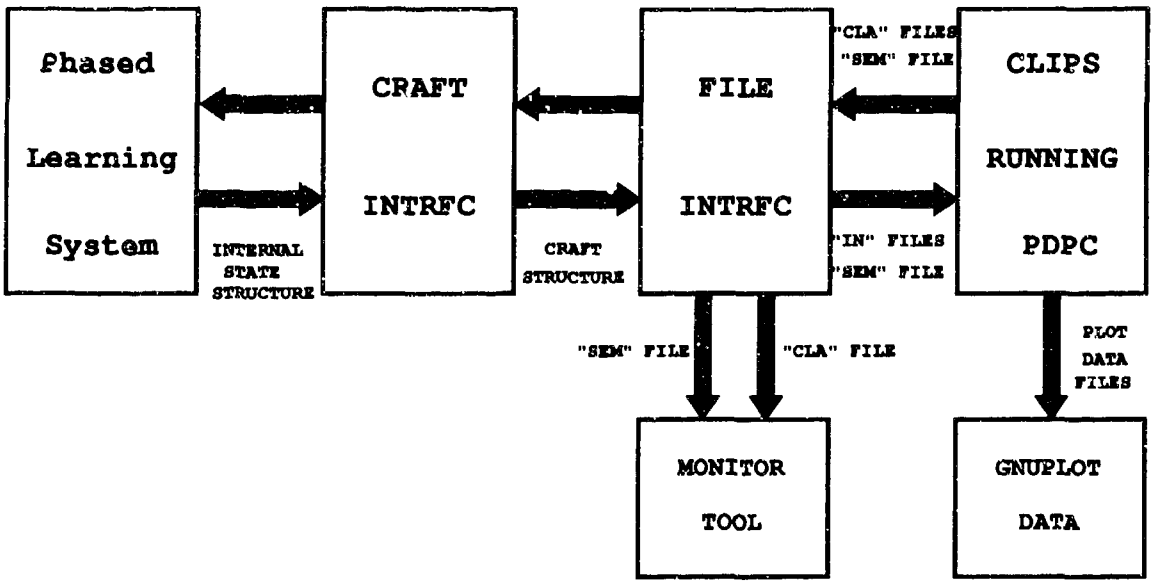


Figure 5.3 Interfacing PPLS to the outside world.

The system interfacing is given in Figure 5.3 below.

The basic PPLS system structure is given in Figure 5.4 below.

Much of the code in the system functions to initialize the system and perform the user interface. The core of the system, the main classifying loop, is a small part of the system. Most of the functionality implement the user interface and the interface layers.

Though the main system uses a functional breakdown, the interfaces are more object-oriented. This was done to facilitate the interfacing of other systems to PPLS with minimum change to the main system. Many other parts of the system lend themselves to object-oriented design (OOD), as can be seen in the functional OOD diagram.

5.5 The Internal Test Environment

This section describes the test environment built in to the PPLS system. This environment allowed for the testing of learning algorithms without the need of loading and operating the Clips-based simulation environment of PDP-C.

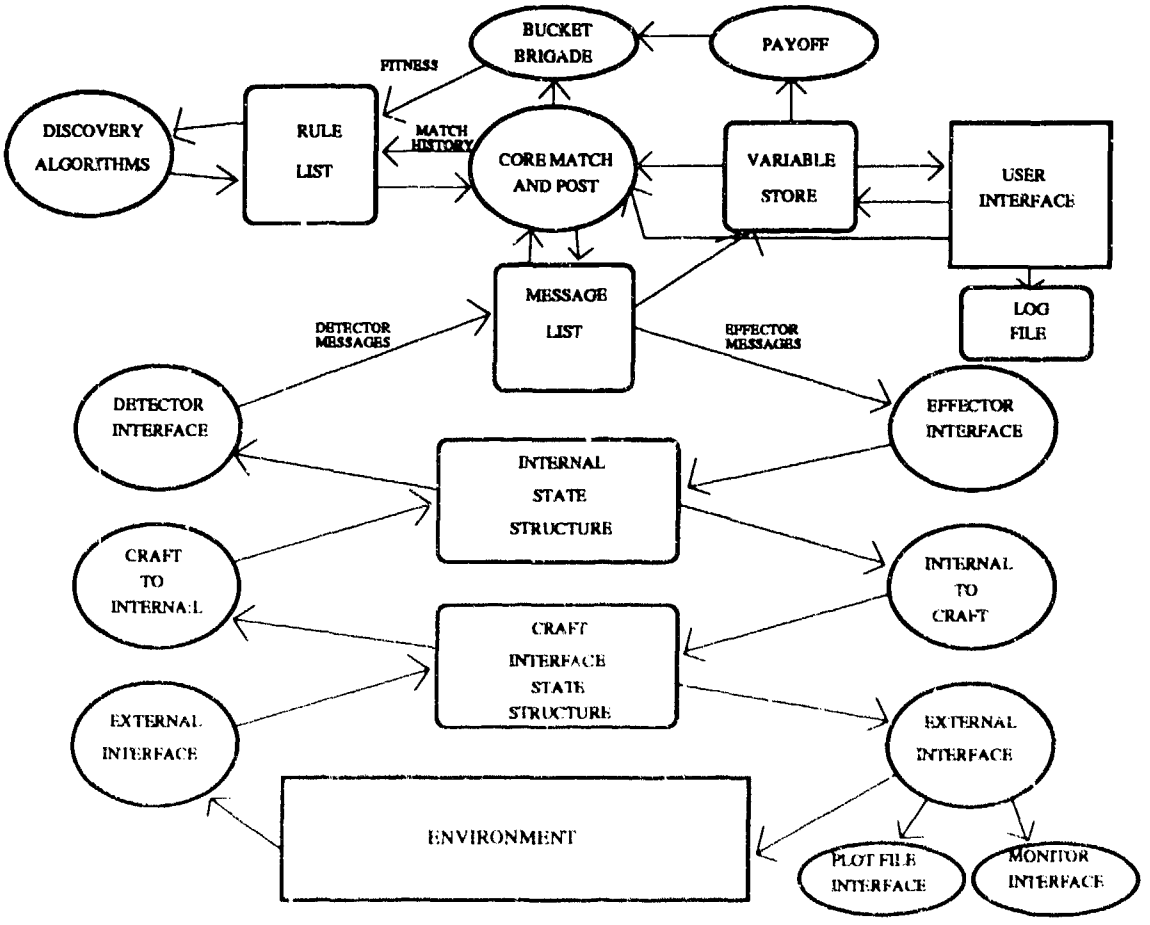


Figure 5.4 PPLS data flow.

5.5.1 Description. The PPLS system includes an interface, called "testenv-intfc.c", that can be linked in to the PPLS system to create a test image called "plst". This test version of PPLS uses many of the same file structures of the external interface system, and so tests many of the features of this interface. But because the system does not require the Clips-based PDP-C system to be loaded and running, the plst system runs much faster.

The interface structure was shown in Figure 5.3. The system uses a set of files to maintain the data links. These files, "testfile.in" and "testfile.cla", mimic the functions of the PDP-C files. "testfile.in" is used to pass data from the PPLS system out to the environment, just as the "edcraft.in" file serves that purpose in the PDP-C system. "testfile.cla" has the same format as the "edcraft.cla" file written by the PDP-C system. This file receives information from the external environment and passes it to the waiting PPLS learning system. In the PDP-C system these two files are implemented as Unix named pipes to minimize file I/O. Here they are actual files to allow the data to be examined.

The file format used in both "testfile.in" and "testfile.cla" is given in Figure 5.5, while the data structures used are given in Figures 5.6 and 5.7.

The following steps are executed every test system interface cycle. These functions are all handled by the `update_external_world` routine in "`testenv-intfc.c`", which is the primary entry point for environment interfacing:

1. New data from the updated state of the environment is written to the "`testfile.cla`"file. In this test version of the system this data is generated by copying the "`testfile.in`"file onto the "`testfile.cla`"file, making state modifications as the copy proceeds. The reading and writing functions are handled by the "`slotio.c`" interface, which is designed to interface to the Clips COOL data structure that PDP-C writes and reads. The update function is handled by the `update_external_craft` routine in the "`testenv-intfc.c`"package.

```
([testcraft] of EDCRAFT
   (name-of edcraft)
   (side neutral)
   (phase Cruise)
   (state moveable)
   (location 0 0 100)
   (velocity 0 0 0)
   (orientation 3.84e-08 -0.11 -0.961)
   (goal waypoint-2)
   (goal-location 50 200 0)
   (desired-direction toward)
   (abc-velocity 10.0 -4.49e-07 -2.06e-07)
   (abc-acceleration -1.325e-05 0 0)
   (abc-thrust -1.325e-05 0 0)
   (abc-attitude -3.84e-08 2.061e-08 -4.491e-08)
   (attitude-rate 0 0 0)
   (attitude-moment 0 0 0)
   (throttle 0.000000)
   (mass 10)
   (on-the-ground FALSE)
   (fuel 8607.769237047467)
   (role leader)
   (leader-of none)
   (follower-of none)
   (mission testing)
   (assignment flight-test)
   (plan neutral-flight-test)
   (condition alive)
   (missile-load 3)
   (type-of fighter)
   (number-of 1)
   (tactical-coordination none)
   (formation none)
   (approach none)
   (bearing-to-defensive-target none)
   (distance-to-defensive-target none)
   (kill-radius-of-defensive-target none)
   (maneuver none)
   (target-name none)
   (target-status alive not_attacked)
)
```

Figure 5.5 Contents of the interface files "testfile.in" and "testfile.cla".

```
/****  CRAFT STRUCTURE  ****/

struct craft_struct {
char_slot self_name;
char_slot self_side;
char_slot self_phase;
char_slot self_state;

xyz_slot self_location;
xyz_slot self_velocity;
orient_slot self_orientation;
orient_slot self_attitude_rate;
double self_throttle;

xyz_slot target_location;

xyz_slot enemy_location;
xyz_slot enemy_velocity;
orient_slot enemy_orientation;

double enemy_attack_range;

xyz_slot base_location;

char_slot goal; /* not currently used */

/* bomb count */
int self_missile_load;
double ordnance_range;
char_slot target_status;
char_slot target_attack_status;
};
typedef struct craft_struct craft_state;
```

Figure 5.6 The craft_state data structure.

```
/* detector bit settings */
int speed; /* SP */
int direction; /* DIR */
int target_aspect_h; /* THO */
int base_aspect_h; /* BHO */
int enemy_aspect_h; /* EHO */
int target_distance; /* DIS */

float target_horiz_offset; /* THO_long */
int over_target; /* OT */
int ordnance_count; /* OC */
float ordnance_range;
float base_horiz_offset; /* BHO_long */
int at_base; /* AB */
float enemy_horiz_offset; /* EHO_long */

/* target status */
int last_target_status;
int target_status; /* TS */
int target_attack_status;

/* distance state vars */
float distance; /* from wheresever going */
float distance_tolerance; /* tolerance on position */
float distance_target_tolerance; /* tolerance on ordnance */

/* mapping from external interface (for internal use to calc above) */
chars self_name;
chars self_side;
chars self_phase;
chars self_state;
chars self_goal;
xyz self_location; /* SL */ /* S = SELF */
xyz self_velocity; /* SV */
orient self_orientation; /* SO */
orient self_attitude_rate; /* SA */
float self_throttle;

xyz target_location; /* TL */ /* T = TARGET */

xyz enemy_location; /* EL */ /* E = ENEMY */
xyz enemy_velocity; /* EV */
orient enemy_orientation; /* EO */
float enemy_attack_range; /* EAR */

xyz base_location; /* BL */ /* B = BASE */

/* outputs */
int  maneuver;
```

Figure 5.7 The internal_state data structure.

2. The updated data is now read in by the `get_craft_data` routine, which converts to an intermediate format held in a `craft_state` structure. Checks are performed on this data, then the `craft_state` structure is passed back to the `update_external_world` routine which, in turn, passes it back to the calling `envir3.c` package that interfaces the environment to the generic `pls.c` main routines package. `envir3.c` calls `convert_craft_to_state` to convert the craft data to the internal form used by the system. This `internal_state` structure is also mapped into the variable space of the system, allowing the user to display and modify most aspects of this interface data.

3. If any action is pending in the system, it is now applied by the routine `apply_action_to_craft` which is called in "`testenv-intfc.c`". This routine applies the lastest action generated by a triggered effector in the main CFSC-1 system. Actions include speed corrections and turn operations. The updates are applied to the internal state structure `current_state` and passed back to `update_external_world`.

4. The internal state data, now updated from the point of view of the PPLS system, is now converted to the `craft_state` structure. Any type conversions are handled at this level. The new structure is then passed back to `update_external_world`.

5. The updated data is then finally passed back to the environment by being written to the "`testfile.in`"file using another routine in the "`slotio.c`" package.

6. The routine `update_external_worldthen` returns to the caller, allowing the classifier system to process the just received data at the same time that the environment reacts to the new data. This creates a one-step delay in the interfacing, but is assumed to be manageable for these test conditions.

5.6 Interfacing to the External Environment

5.6.1 The Pilot Decision Phases in C simulation system.

5.6.2 Interfacing to the PDP-C simulation. The interfacing here is similar to that used in the internal test environment. For clarity, the steps used are elaborated fully below. As before, all steps are coordinated by the `update_external_world` routine, which provides the single interface from PPLS to the environment.

1. If this is the first access to the PDP-C system, set the flag in "`edcraft.sem`"to the ASCII text "MAKE-EDCRAFT". This forces the PDP-C simulation to create the edcraft object and create the "`edcraft.cla`"file. This is only done once.

2. New data from the updated state of the environment is written to the "`edcraft.cla`"file. "`edcraft.cla`"is implemented as a Unix named pipe, so system execution stops until an end-of-file is written to the pipe by the PDP-C system. The file "`edcraft.sem`"received the value "FALSE", indicating that the data in the file "`edcraft.in`"is now considered old.

3. The updated data is now read in by the `get_craft_data` routine, which converts to an intermediate format held in a `craft_state` structure. This step is similar to the same step in the internal interface operation. Checks are performed on this data, then the `craft_state` structure is passed back to the `update_external_world` routine which, in turn, passes it back to the calling `envir3.c` package that interfaces the environment to the generic `pls.c` main routines package. `envir3.c` calls `convert_craft_to_state` to convert the craft data to the internal form used by the system. This `internal_state` structure is also mapped into the variable space of the system, allowing the user to display and modify most aspects of this interface data.

4. If any action is pending in the system, it is now applied by the routine `apply_action_to_craft` which is called in "`edcraft-intfc.c`". This routine applies the lastest action generated by a triggered effector in the main CFSC-1 system. Actions include speed corrections and turn operations. The updates are applied to the internal state structure `current_state` and passed back to `update_external_world`.

5. The internal state data, now updated from the point of view of the PPLS system, is now converted to the craft_state structure. Any type conversions are handled at this level. The new structure is then passed back to update_external_world.

6. The updated data is then finally passed back to the environment by being written to the "edcraft.in"file using another routine in the "slotio.c" package. The semaphore file "edcraft.sem"is modified to contain the ASCII text "TRUE", telling the PDP-C system that the data in the "edcraft.in"file is now valid.

7. The routine update_external_world then returns to the caller, allowing the classifier system to process the just received data at the same time that the environment reacts to the new data. This creates a one-step delay in the interfacing, but is assumed to be manageable for these test conditions.

The PDP-C system reacts to the updated object record by reading select fields from this structure and using rule firings to move the craft. The edcraft object has been designed as an externally controllable agent in the PDP-C system and is immune to many system control rules. A balance was created between object control and object simulation, allowing the system to move (according to the move data in the "edcraft.in"file) the craft and other agents to react to this agent, but to prevent the simulation from taking control of it.

The PDP-C system is partially connected to the DIS network at this time, allowing the actions of the system to be monitored by a DIS graphical monitoring interface. This provides a good "view" of the activities between the four agents and the fifth edcraft agent. Data is currently passed to the network via data files and a conversion program, and so is not yet real time. Plans are under way to introduce DIS objects received from the network into the PDP-C simulation, possibly as additional edcraft objects. This will allow complete DIS interaction, within the limits of the PDP-C simulation.

5.7 Summary and Discussion

The design presented in this chapter for the Phased Pilot Learning System is a complex web of simpler objects and networking interfaces. This design allows each of the individual components to function autonomously via loosely coupled interfaces, which allows it to be distributed between different processes or even different hardware.

One of the main stumbling blocks was the CFSC-1 source code itself. The code was not that well documented in parts and many of the interface structures not completely obvious. Making modifications to this code had to be done cautiously to avoid side effects that effected the rest of the system's operation. That said, the modular design of the CFSC-1 system allowed the PPLS system to be quickly implemented and the advanced user features, both provided by CFSC-1 and added by the author, made the monitoring of the binary rule system realatively easy.

VI. Empirical Results

This chapter presents the tests performed on the PPLS system and the results. First, we present the test format, noting its complexity within the search domain of the environment. Three basic forms of this test are used. The first form loads a set of rules into the PPLS classifier system and demonstrates that the phasing approach can be used to control the agent's behavior appropriately. This test shows that the agent controller can act effectively as a standard rule processing system to control the agent using rote (implanted) knowledge and deductive information filtering.

Then some of the rules are removed from the rule set, leaving gaps in behavior that must be overcome. This represents the case where most of the behaviors of the agent are useful, but where no specific rules address a specific agent need. The agent must use discovery learning techniques to induce the needed rules. Other rules in the population can function as *seed rules* to guide this discovery process as the system uses a form of analogy to build new rules from parts of these seed rules via the genetic algorithm.

Last, the rule population is initialized with randomly-generated rules (except for the phasing rules) and tests are run to see how the agent adapts using the phasing rules and rewards as guidance. This situation forces the agent to rely fully on inductive learning (at first) to generate the needed behaviors. This is a rather drastic situation, similar to throwing someone off the street into an aircraft cockpit, but provides a base for later comparisons with the mixed strategy learning scenarios provided by rule seeding.

The tests are carried out on the relatively static internal environment. As other researchers have shown, even a small amount of knowledge implantation into the rule base can significantly affect the performance of the discovery algorithms.

5.7 Summary and Discussion

The design presented in this chapter for the Phased Pilot Learning System is a complex web of simpler objects and networking interfaces. This design allows each of the individual components to function autonomously via loosely coupled interfaces, which allows it to be distributed between different processes or even different hardware.

One of the main stumbling blocks was the CFSC-1 source code itself. The code was not that well documented in parts and many of the interface structures not completely obvious. Making modifications to this code had to be done cautiously to avoid side effects that effected the rest of the system's operation. That said, the modular design of the CFSC-1 system allowed the PPLS system to be quickly implemented and the advanced user features, both provided by CFSC-1 and added by the author, made the monitoring of the binary rule system realatively easy.

VI. Empirical Results

This chapter presents the tests performed on the PPLS system and the results. First, we present the test format, noting its complexity within the search domain of the environment. Three basic forms of this test are used. The first form loads a set of rules into the PPLS classifier system and demonstrates that the phasing approach can be used to control the agent's behavior appropriately. This test shows that the agent controller can act effectively as a standard rule processing system to control the agent using rote (implanted) knowledge and deductive information filtering.

Then some of the rules are removed from the rule set, leaving gaps in behavior that must be overcome. This represents the case where most of the behaviors of the agent are useful, but where no specific rules address a specific agent need. The agent must use discovery learning techniques to induce the needed rules. Other rules in the population can function as *seed rules* to guide this discovery process as the system uses a form of analogy to build new rules from parts of these seed rules via the genetic algorithm.

Last, the rule population is initialised with randomly-generated rules (except for the phasing rules) and tests are run to see how the agent adapts using the phasing rules and rewards as guidance. This situation forces the agent to rely fully on inductive learning (at first) to generate the needed behaviors. This is a rather drastic situation, similar to throwing someone off the street into an aircraft cockpit, but provides a base for later comparisons with the mixed strategy learning scenarios provided by rule seeding.

The tests are carried out on the relatively static internal environment. As other researchers have shown, even a small amount of knowledge implantation into the rule base can significantly affect the performance of the discovery algorithms.

```
;
; A Simple Multi-phase operation
;
; Inputs: Y = Classifier Type, Ph = Phase, NP = Next Phase (via PHASE cfop), Tg = Tag, THO = Target Horizont
;      OT = Over Target, OC = Ordnance Count, sp = Speed, BHO = Base Horizontal Offset, AB = At Base, DIS = D
; Outputs: Man = Maneuver
;
; OP = Operation rule performs:  none = Pass through, PHASE = change phase of system, STOP = stop a classify
;
; Params 9999 = set strength of rule to 9999
;

;Phase 0 - fly to target
; steering
if (ty det)(ph 0)  (tho= left)(ot no)(sp slow) and () then (ty eff)(man left)
if (ty det)(ph 0)  (tho= left)(ot no)(sp cruise) and () then (ty eff)(man left)
if (ty det)(ph 0)  (tho= right)(ot no)(sp slow) and () then (ty eff)(man right)
if (ty det)(ph 0)  (tho= right)(ot no)(sp cruise) and () then (ty eff)(man right)
; speed
if (ty det)(ph 0)  (sp stopped) and () then (ty eff)(man accel)
if (ty det)(ph 0)  (sp slow)(dis FAR) and () then (ty eff)(man accel)
if (ty det)(ph 0)  (sp slow)(dis Medium) and () then (ty eff)(man accel)
if (ty det)(ph 0)  (sp    WISE)(dis CLOSE) and () then (ty eff)(man SLOW)
if (ty det)(ph 0)  (sp     T)(dis CLOSE) and () then (ty eff)(man SLOW)
if (ty det)(ph 0)  (oc 1) (ot yes) and () then (ty eff)(man drop)
if (ty det)(ph 0)  (oc 2) (ot yes) and () then (ty eff)(man drop)
if (ty det)(ph 0)  (oc 3) (ot yes) and () then (ty eff)(man drop)
; Phase change
if (ty det)(ph 0)  (tho 0) and () then (ty phase) (np 1)(man none) op PHASE params 9999
if (ty det)(ph 0)  (oc 0) and () then (ty phase) (np 1)(man none) op PHASE params 9999
;
;Phase 1 - fly back to base
if (ty det)(ph 1)  (ab no)(bho left) and () then (ty eff)(man left)
if (ty det)(ph 1)  (ab no)(bho right) and () then (ty eff)(man right)
if (ty det)(ph 1)  (ab yes) and () then (ty phase)(np 2) op PHASE params 9999
;
;Phase 2 - done
if (ty det)(ph 2)   then (ty phase) op STOP params 9999
;
;Phase 4 - evade enemy
if (ty det)(ph 0)(eho 1) and () then (ty phase) (np 4)(man none) op PHASE params 9999
if (ty det)(ph 0)(eho 2) and () then (ty phase) (np 4)(man none) op PHASE params 9999
if (ty det)(ph 4)(eho 0) and () then (ty phase) (np 0)(man none) op PHASE params 9999
; steering
if (ty det)(ph 4)  (eho= left)(sp slow) and () then (ty eff)(man right)
if (ty det)(ph 4)  (eho= left)(sp cruise) and () then (ty eff)(man right)
if (ty det)(ph 4)  (eho= right)(sp slow) and () then (ty eff)(man left)
if (ty det)(ph 4)  (eho= right)(sp cruise) and () then (ty eff)(man left)
```

Figure 6.1 The rules used to test execution without learning.

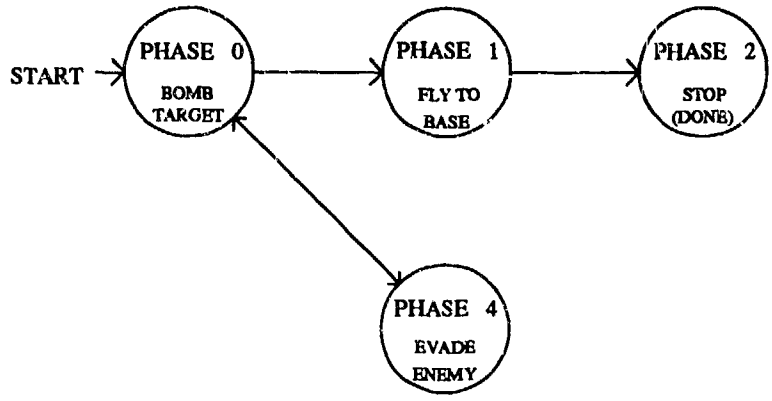


Figure 6.2 Sequence of events in test problem.

This also implements default hierarchies, as is described in Appendix B. This can be seen by the use of non-specified condition fields (that do not show up in the rules) that allow some rules to match more general conditions than others.

For this test the learning mechanisms were turned off, so no changes in the rule list are made during execution. The Bucket Brigade credit allocation mechanism, however, remains on in order to update the fitnesses of the rules. This should allow the fitnesses to tend to their fixed-point values for this set of tasks(59).

The test starts with the system in Phase 0 (which is always the starting phase). The agent begins to progress forward by executing rules that excelerate the aircraft. Once some speed has been attained, the agent steers to the first destination, the target on which to drop its ordnance.

The agent then is informed of an enemy craft in its path. This state detection forces the learning system into Phase 4, which contains rules that allow the agent to evade the enemy. The agent stays in Phase 4 until the enemy is no longer detected (designed to occur when the agent is a specific distance from the enemy). This is communicated to the agent via the THO (target horizontal field value "-" (00) of this field.

Once the agent is close enough to the target, it drops an ordnance. Hitting the target switches the system (via rule firings) to Phase 1. This phase concentrates on returning to the agent's base.

Once at the base, the agent switches to Phase 2 (the STOP phase, since the only rule in this phase activates the STOP action) and the execution of the system stops. At this point the simulation can be changed to another phase and continued (based on the condition matches of the rules in the system). The enemy avoidance rules can be added to Phase 1 also, though this was not done to keep the rule set simple.

Figure 6.4 shows the path the agent took in one test run (other runs are similar, but vary due to probabilistic selection of actions[3]). This plot demonstrates that navigation and control using a rule-based classifier system is possible.

This test shows that implanted knowledge can be used effectively to guide an agent though a set of tasks. This is useful since many tasks faced by the learning system will have "cook book" approaches that can be effectively implanted into the agent. A guiding philosophy here can be to limit learning to only those things that must be learned. This use of rote learning (as defined at the beginning of Chapter 2) provides a way to make the agent's learning more efficient in the task domain. This is similar to how Soar(67) uses chunks to solve problems it has seen before. The question addressed next is whether this form of learning is compatible with the discovery-based learning methods of the next section.

6.2 Internal Tests With Incremental Learning

Incremental learning, as applied here, refers to adding to an existing base of knowledge in an incremental way that does not seriously affect the ability of the system to perform. This is an especially useful form of learning when an on-line agent encounters a situation that it has not learned how to handle yet. In this context, we are interested in adding or modifying rules in the existing rule base when a situation arises not currently covered by the existing set of rules.

[3] The reader is referred to the test results appendix and the settings of the various variables used to define the operation of PPLS. Here we refer to the setting of the Effector Resolution Mechanism flag (eff_res mech) that was set to 2, which instructs the system to choose the action that is most supported by the bidding rules, but on a probabilistic basis. Thus rules may fire that are not the best supported, but only rarely.

```
129  CfStrMax, max_cf_str, 9999 "max cf strength"
130  CfStrMin, min_cf_str, 1 "min cf strength"
131  CfBidMin, min_cf_bid, 0.1 "min cf bid"

201  OneMPerC, one_msg_per_cf, 1 "limit each cf to 1 msg"
202  DelHall, no_hall, 1 "no internal DET msgs"
206  DetectRt, detector_rate, 1 "how often dets sampled"
207  EffectRt, effector_rate, 1 "activate matched effs this often"
244  Bid_k, bidk, .1 "bid risk factor"
252  DMShare, bba_frac_det_msg_share, 1 "frac det msg gets rel to cfs (def  1.0)"
253  FrPayNEC, bba_frac_pay_non_eff, 1 "frac pay non-eff activating cfs (def 1.0)"

301  HeadTax, tax_head_rel, .0005 "sub str*head_tax each round"
302  AHeadTax, tax_head_abs, 0 "sub aheadtax each round"
303  BidTax, tax_for_bidding .005
304  PrdTaxMx, tax_produc_max, 0 "production tax"
306  FBidTax, tax_failed_bid, 0 "tax on cf if bid but not produce"

619 MinNewCfs, min_new_cfs, 2 "if > 0, is min num cfs to create (see FrNewCfs)"
621 FrNewCfs, frac_new_cfs, .04
"frac of cf pop repl w/ new cfs = NmCfs*FrNewCfs (see MinNewCfs)"
622 CrowdFac, repl_crowding_factor, 1
"1=pick 1 rule to repl;>1=pick many, of those repl most like"
625 PkPrnWOR, disc_pick_par_no_repl, 0
"1=pick rule only once as parent (w/o replacement); 0=no limit"
628 RandRpic, repl_how_pick_replacement, 0
"0=inv str;1=eq prob;2=1 w/ RplCfUBd, RplACSBd, RplACUBd limits"
631 MxCfCopy, repl_max_cf_copy, 0
"max number of identical rules; 0=no limit; >0=do slow check"
633 BkgGARt, ga_bkgnd_ga_rate, .1 "Prob that GA will be used in a cycle; 0=no GA"
637 BGABPPr, disc_bidding_par_prob, 0.
"pick bidding parents only prob (0.5 -> half time start w/ bdrs)"
700 MuPrTot, mut_prob_total, .04 "total prob of mutating a cf"
701 MuFrNSL, mut_frac_nal, .2 "prob mutate loci to wild"
720 CDMsgs, cover_det_on, 1 "1=on"
721 CDMsgsRt, cover_det_op_rate .1
800 CEffs, cover_eff_on, 1
"0=off;1=make 1 cf when triggered;2=make 2 cfs; (see eff struct)"
801 CEffsRt, cover_eff_prob, .1
"prob that activate Cov efi op when triggered (by MadsMtsk)"

973 phase_on, Using_Phase, 1 "turn use of phase on"
974 phase_pars, Parents_Within_Phase, 0 "choose parents from same phase"
975 phase_repl, Replace_Within_Phase, 1 "replace only rules in same phase"
979 eff_res_mech, env_eff_res_mech, 3
"0=use high bid;1=use highest supported;2=support as prob"
983 PhaseCfMax, Max_Cfs_per_phase, 40 "Max number of Cfs in rule list for one phase"
989 penact, penalize_nonaction, 1 "1 = on;  no action = mistake"
990 genwild2, generate_wild_cond2, 1 "1 = on: when gen random cfs, make cond2 all wild"
992 expop, population_expansion, 1 "1 = allow pop to expand to NmCfsMx (or Phase limit)"
994 covbada, cover_det2_ch_act_prob, .1
"prob that Mistake triggered det cover will mod action"
```

Figure 6.3 Some of the variable settings used.

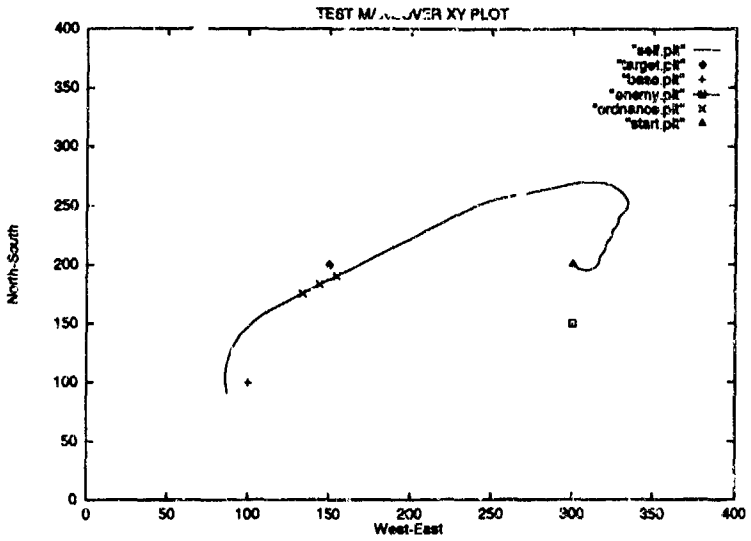


Figure 6.4 The path taken by the agent: implanted rules in the test environment.

We test PPLS's ability to incrementally learn by loading a partial set of rules and placing the agent into a similar, but unlearned situation. The goal is to have the system add rules to the rule base that compensate for the unlearned environmental concepts. Specifically, we test the agent's ability to adjust to learn rules to return the craft to base.

In these tests the following set of rewards were used. Rewards are paid to the learning system when a trigger condition is met at the end of a classifier cycle. The reward values for these tests are from the following values:

Better Reward — Paid out when the state of the system has improved (progress to the current phase objectives is made).

Worse Reward — Paid out when the state of the system is worse (generally a negative value).

Mission Reward — Paid out when an objective of the current phase is met.

Death Reward — Paid out if the agent is killed somehow. (Generally negative.)

The values used in this research were chosen empirically and are listed below. These values are not optimised and better values probably exist.

Better Reward	50
Worse Reward	-50
Mission Reward	1000
Death Reward	-1000

The actual reward situations are listed by the phase they occur. Rewards applicable to all phases are listed last.

Phase 0	Better Reward	The agent has moved closer to the enemy base (target).
	Worse Reward	The agent has moved away from the target.
	Mission Reward	The agent has destroyed the target.
Phase 1	Better Reward	The agent has moved closer to its base.
	Worse Reward	The agent has moved farther away from its base.
	Mission Reward	The agent is over its base.
Phase 4	Better Reward	The agent has moved *away* from the enemy.
	Worse Reward	The agent has moved *closer* to the enemy.
	Mission Reward	The enemy is no longer detected (out of range).
All Phases	Death Reward	The agent is killed (by the enemy).

The rewards provide the learning algorithm with guidance on the usefulness of its decisions and form part of the implanted knowledge that describes a mission to the agent. Also set by the reward function is the variable `MadeMatk`. This is set whenever progress has been negative over a period of time to trigger the discovery learning algorithms (if turned on).

The following tests are executed:

1 - A complete[4] set of rules is loaded into the system. Learning is turned on and the effects of the learning algorithm on the stability of the rule base is examined over a series of runs.

2 The rule set that evolved from Test 1 is tried in other situations where the starting position, base locations, and enemy location are moved. This checks the adaptibility of the rule set and the

[4]In the sense that the rule set can effectively control the agent through the various phases to an efficient arrival at the final destination, the friendly base. The rule set is not considered optimum, but just empirically sufficient (sufficient in most situations tested).

system, i.e. sees if the system with this set of rules is *brittle* (fails outside of the context the rules were learned in or designed for).

3 Some of the rules in the rule set are removed and the ability of the system to compensate examined. The discovery learning algorithms are to create rules that allow the agent to reach its objectives. For this test, only some rules in one phase (the Return to Base phase, Phase 1), are removed.

4 All the rules in one phase are removed. Phase 1 is again used for the test.

5 A random set of rules are loaded into the system and its ability to navigate in the environment examined.

The discovery learning algorithms used by PPLS follow.

Detector Cover – This operator triggers when either no rule matches and input detector message. I have modified the operator to also trigger when a mistake has been made by the system, such as when a series of negative rewards has been received. The operator generates a new rule that matches the input (with some of the condition fields generalized randomly) and has a randomly generated action. The new rule is set to have a fitness value equal to the current population average. The new rule may be applied (probabilistically) the next time this input is seen.

Effector Cover – This operator triggers when a mistake is indicated by the reward system (as above) and a rule matched the input. It generates a rule that has the same condition fields but has a random action. The idea is that the previous rule might be correctly matching the input detector message (firing in the right situation), but is not generating an action appropriate to that situation.

Genetic Algorithm – The genetic algorithm triggers at a set background rate, determined probabilistically. When triggered, it selects two rules from the *parents pool* (a pool of rules that

are high fitness and meet any other requirements set by the system variables, such as being within the current phase) and applies the genetic operators of crossover and mutation with a probability set by the system variables. A background rate of 0.1 (meaning that the operator should fire on average once every tenth cycle) was empirically chosen based on information on CFSC(59) and used in the tests.

In the first test of this type, we begin with a complete starting set of rules and, with learning on (see below), execute the scenario from start to end. We then use the rule set as it was modified by the system and repeat the test from the beginning. The intent of this test is to validate that the system can perform correctly over many activations with an evolving rule set.

The initial rule set (in symbolic form) is depicted in Figure 6.5.

The results are presented in Figures 6.6 6.7, and 6.8. The first figure shows the system on the first execution. The second execution of the mission scenario is shown in the second figure which shows that the route has been modified to increase the overall fitness of the route. The third plot shows that learning has reached a (temporary) steady state under these conditions.

Next the evolved rule set was placed in a different situation and executed. The results of the first four runs is shown in Figures 6.9, 6.10, 6.11, and 6.12.

After this second set of tests, the rule set evolved to that shown in Figure 6.13.

The tests show that a rule set can effectively evolve while maintaining a level of performance. This is important if the discovery algorithms (the collection of algorithms that implement rule discovery) are to be left on during system operation.

The reader is referred to the Appendix for other test results.

The results show that a partial seeding allows the system to discover a useful set of rules after some effort. This ability to discover appropriate rules using past experience (other rules) and

```
;
; A Simple Multi-phase operation
;
; Inputs: Y = Classifier Type, Ph = Phase, NP = Next Phase (via PHASE cfop), Tg = Tag, THO = Target Horizonta
;      OC = Ordnance Count, sp = Speed, BHO = Base Horizontal Offset, AB = At Base
; Outputs: Man = Maneuver
;

;Phase 0 - fly to target and destroy it
; steering
if (ty det)(ph 0)  (tho= left)(ot no)(sp slow) and () then (ty eff)(man left)
if (ty det)(ph 0)  (tho= left)(ot no)(sp cruise) and () then (ty eff)(man left)
if (ty det)(ph 0)  (tho= right)(ot no)(sp slow) and () then (ty eff)(man right)
if (ty det)(ph 0)  (tho= right)(ot no)(sp cruise) and () then (ty eff)(man right)
if (ty det)(ph 0)  (tho= front)(ot no)(sp cruise) and () then (ty eff)(man straight)
if (ty det)(ph 0)  (tho= front)(ot no)(sp slow) and () then (ty eff)(man straight)
; speed
if (ty det)(ph 0)  (sp stopped) and () then (ty eff)(man accel)
if (ty det)(ph 0)  (sp slow)(dis FAR) and () then (ty eff)(man accel)
if (ty det)(ph 0)  (sp slow)(dis Medium) and () then (ty eff)(man accel)
if (ty det)(ph 0)  (sp CRUISE)(dis CLOSE) and () then (ty eff)(man SLOW)
if (ty det)(ph 0)  (sp FAST)(dis CLOSE) and () then (ty eff)(man SLOW)
if (ty det)(ph 0)  (oc 1) (ot yes) and () then (ty eff)(man drop)
if (ty det)(ph 0)  (oc 2) (ot yes) and () then (ty eff)(man drop)
if (ty det)(ph 0)  (oc 3) (ot yes) and () then (ty eff)(man drop)
; Phase change
if (ty det)(ph 0)  (tho 0) and () then (ty phase) (np 1)(man none) op PHASE params 9999
if (ty det)(ph 0)  (oc 0) and () then (ty phase) (np 1)(man none) op PHASE params 9999
;
;Phase 1 - fly back to base
if (ty det)(ph 1)  (ab no)(bho left) and () then (ty eff)(man left)
if (ty det)(ph 1)  (ab no)(bho right) and () then (ty eff)(man right)
if (ty det)(ph 1)  (ab no)(bho front) and () then (ty eff)(man straight)
if (ty det)(ph 1)  (ab yes) and () then (ty phase)(np 2) op PHASE params 9999
if (ty det)(ph 1)  (sp CRUISE)(dis CLOSE) and () then (ty eff)(man SLOW)
if (ty det)(ph 1)  (sp FAST)(dis CLOSE) and () then (ty eff)(man SLOW)
;
;Phase 2 - done
if (ty det)(ph 2)   then (ty phase) op STOP params 9999
;
;Phase 4 - evade enemy
if (ty det)(ph 0)(eho 1) and () then (ty phase) (np 4)(man none) op PHASE params 9999
if (ty det)(ph 0)(eho 2) and () then (ty phase) (np 4)(man none) op PHASE params 9999
if (ty det)(ph 4)(eho 0) and () then (ty phase) (np 0)(man none) op PHASE params 9999
; steering
if (ty det)(ph 4)  (eho= left)(sp slow) and () then (ty eff)(man right)
if (ty det)(ph 4)  (eho= left)(sp cruise) and () then (ty eff)(man right)
if (ty det)(ph 4)  (eho= right)(sp slow) and () then (ty eff)(man left)
if (ty det)(ph 4)  (eho= right)(sp cruise) and () then (ty eff)(man left)
```

Figure 6.5 The rules used to test execution with learning starting with useful implanted rules.

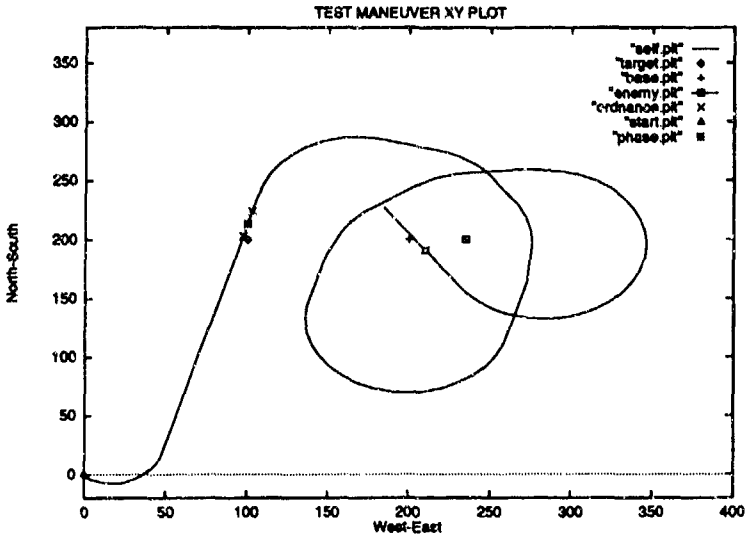


Figure 6.6 The path taken by the agent: First run.

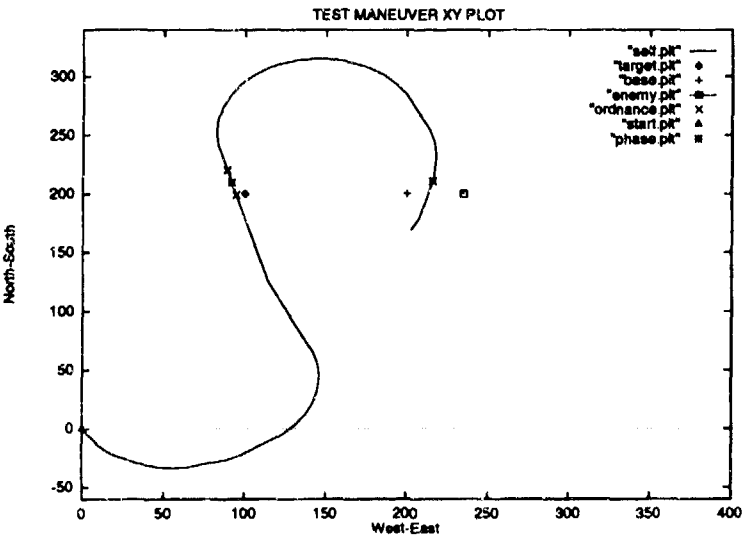


Figure 6.7 The path taken by the agent: Second run.

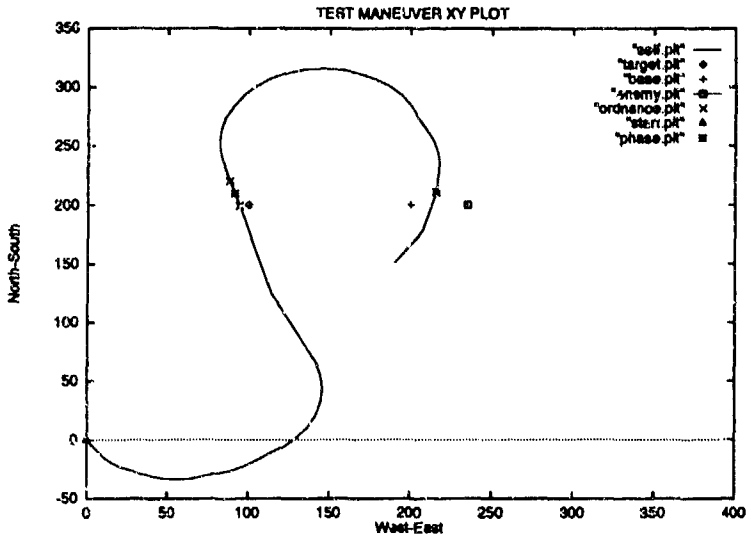


Figure 6.8 The path taken by the agent: Third run.

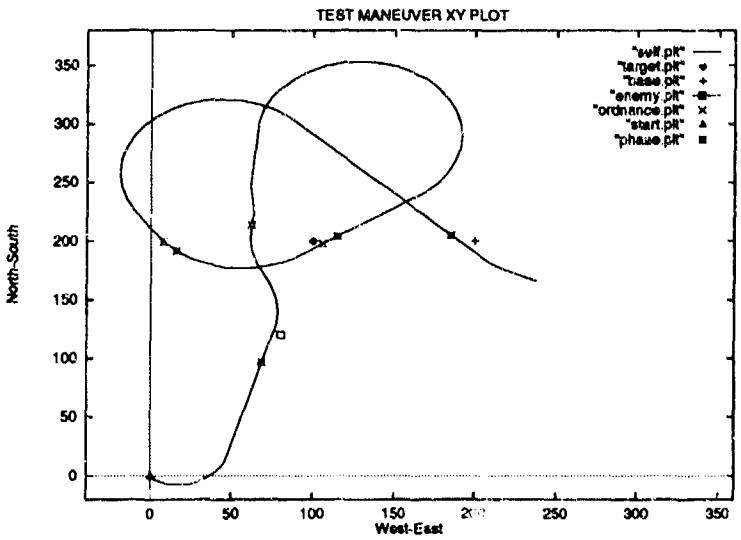


Figure 6.9 Different situation test: First run.

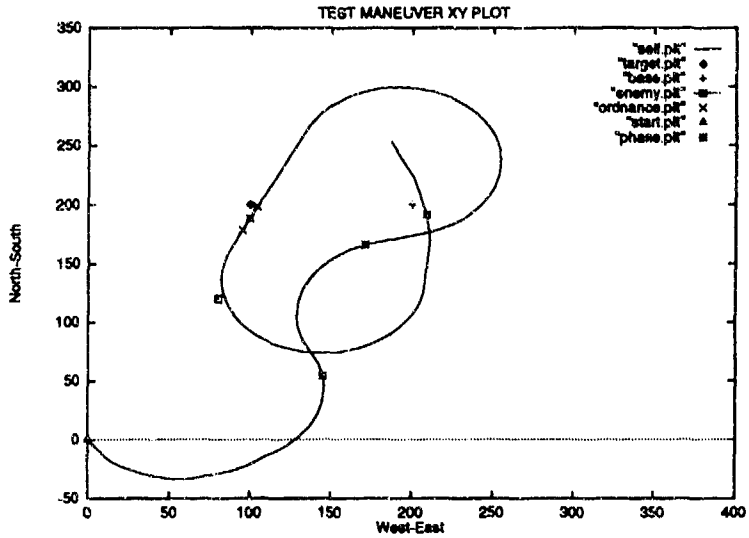


Figure 6.10 Different situation test: Second run.

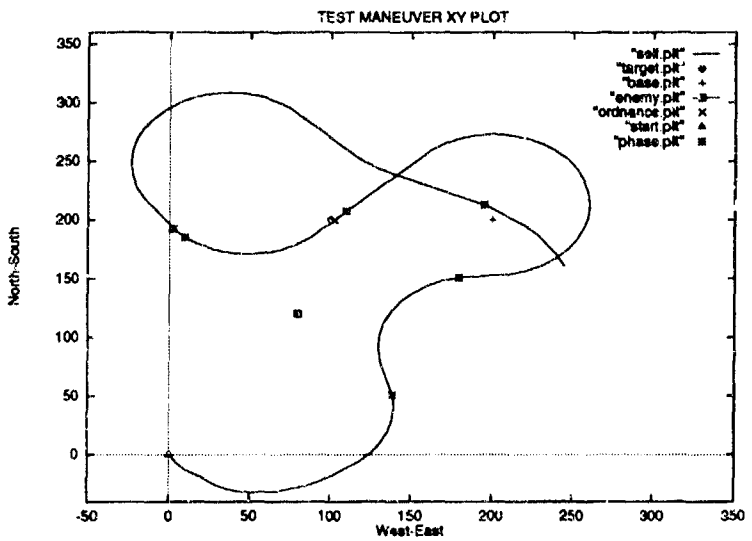


Figure 6.11 Different situation test: Third run.

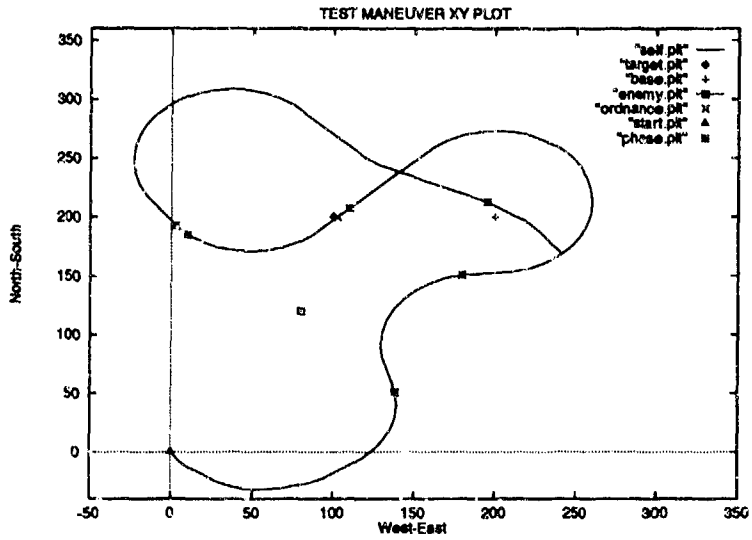


Figure 6.12 Different situation test: Fourth run.

```
20> S 117  0 (TY Det)(PH 1)(AB Yes) and () then (TY Phase)(PH 1)(NP 2)(AB Yes) op PHASE                -str 7255, br 0.13, supt 0.0"
26> S  88  0 (TY Det)(PH 4)(EHO -) and () then (TY Phase)(PH 4)(NP 0)(EHO -)(MAN NONE) op PHASE       -str 6827, br 0.12, supt 0.0"
28>    128  0 (TY Det)(PH 3) and () then (TY Phase)(PH 2) op STOP                                    -str 5481, br 0.09, supt 0.0"
31>   40 108 (TY Det)(PH 4)(EHO Rt)(SP Cruis) and () then (TY Eff)(PH 2)(EHO Rt)(SP Cruis)(MAN STRAIGHT)  -str 4811, br 0.16, supt 0.0"
19>  112  0 (TY Det)(PH 1)(EHO FRONT)(AB No) and () then (TY Eff)(PH 1)(EHO FRONT)(AB No)(MAN STRAIGHT)  -str 3635, br 0.16, supt 0.0"
21>  116  0 (TY Det)(PH 1)(SP Cruis)(DIS Close) and () then (TY Eff)(PH 1)(SP Cruis)(DIS Close)(MAN SLOW)  -str 2627, br 0.16, supt 0.0"
30>   40  0 (TY Det)(PH 4)(EHO Rt)(SP Cruis) and () then (TY Eff)(PH 4)(EHO Rt)(SP Cruis)(MAN LEFT)   -str 2467, br 0.16, supt 0.0"
28>   82  0 (TY Det)(PH 4)(EHO Lt)(SP Cruis) and () then (TY Eff)(PH 4)(EHO Lt)(SP Cruis)(MAN RIGHT)  -str 2462, br 0.16, supt 0.0"
18>  116  0 (TY Det)(PH 1)(EHO Right)(AB No) and () then (TY Eff)(PH 1)(EHO Right)(AB No)(MAN RIGHT)  -str 1686, br 0.16, supt 0.0"
17>  108  0 (TY Det)(PH 1)(EHO Left)(AB No) and () then (TY Eff)(PH 1)(EHO Left)(AB No)(MAN LEFT)     -str 1639, br 0.16, supt 0.0"
2>    68  0 (TY Det)(PH 0)(THO LEFT)(OT No)(SP Cruis) and () then (TY Eff)(PH 0)(THO LEFT)(OT No)(SP Cruis)(MAN LEFT)-str 1157, br 0.19, supt 0.0"
27>    0  0 (TY Det)(PH 4)(EHO Lt)(SP Slow) and () then (TY Eff)(PH 4)(EHO Lt)(SP Slow)(MAN RIGHT)    -str 808, br 0.16, supt 0.0"
3>     0  0 (TY Det)(PH 0)(THO RT)(OT No)(SP Slow) and () then (TY Eff)(PH 0)(THO RT)(OT No)(SP Slow)(MAN RIGHT)  -str 784, br 0.19, supt 0.0"
22>    0  0 (TY Det)(PH 1)(SP Fast)(DIS Close) and () then (TY Eff)(PH 1)(SP Fast)(DIS Close)(MAN SLOW)  -str 784, br 0.16, supt 0.0"
29>    0  0 (TY Det)(PH 4)(EHO Rt)(SP Slow) and () then (TY Eff)(PH 4)(EHO Rt)(SP Slow)(MAN LEFT)    -str 784, br 0.16, supt 0.0"
1>    21  0 (TY Det)(PH 0)(THO LEFT)(OT No)(SP Slow) and () then (TY Eff)(PH 0)(THO LEFT)(OT No)(SP Slow)(MAN LEFT)  -str 635, br 0.19, supt 0.0"
36>  108 (108  0) 124 (TY Det)(PH 1)(EHO Left)(AB No) and () then (TY Eff)(PH 2)(EHO Left)(AB No)(MAN RIGHT)  -str 1  br 0.20, supt 0.0"
```

Figure 6.13 The rules in the system after the second set of tests.

a set of heuristics (the discovery algorithms) is an important advantage of genetics-based learning systems in changing environments.

When a random set of rules (or no rules) is used as the starting point, the agent is mostly at the mercy of the discovery algorithms. In these cases the results were not as promising, with the learning system taking more than 1000 cycles to converge, if at all. This is contrasted to the 100 to 200 cycles for the previous tests.

The results indicate that the better (more useful) the starting rules used to seed the initial rule set, the better the performance with these discovery algorithms. Better discovery algorithm directly influence this performance, since they narrow the search heuristically for useful rules.

It was found that allowing the discovery operators the ability to create and modify the phasing rules lead quickly to the system creating rules that lead the system down blind alleys without easy recovery. The STOP rule action also posed a similar problem, since the system would quickly learn to stop itself and avoid any further work. Therefore rules using these rule actions are treated "specially" in that they cannot be deleted from the rule population and they cannot serve as parents for the cover operators. We treat these rules as mission-dependent goals implanted into the system, rather than execution instructions to be learned and used. With some restrictions, however, it should be possible to set up a system that can learn the phases the agent needs to interact with its environment. We elaborate on this in Section 6.4.

The specific discovery operators used in this task were the detector cover operator of CFSC-1, a modified version of the effector cover operator in CFSC-1, and the genetic algorithm. The *cover detector operator* looks for situations where no rule is addressing a current detector input. It then copies and modifies a existing rule so that the condition field of this new rule matches the input detector message and so fires. The action is usually inappropriate, but the genetic algorithm can use this new raw genetic material to build more useful rules that address the situation. The *cover effector operator* is triggered whenever the variable `Madeastk` (made mistake) is set in the

reward routine. In PPLS this is done whenever no action is generated by the learning system. This assumes that some action available to the agent, even flying straight ahead, is always appropriate. The operator works by copying a rule that fired during the current cycle (such as one generated by the cover detector operator) and modifying the action of the rule to form a valid effector control message. In PPLS, this routine was modified to concentrate on actions that are allowed by the interface, removing wildcard bits from these randomly selected trial actions since they are ignored by the interface. The idea is to reduce the complexity of the interface level to as simple a level as possible. The modified operator also copied the majority of the state bits from the condition of the rule, unlike the CFSC-1 version that generated totally random actions. The *genetic algorithm* functions similarly to the standard version in Appendix A, crossing and mutating the bits of the rule strings. The genetic algorithm provides the main inductive mechanism in the system, though the cover operators provide a limited amount of generalization (changing random bits to wildcards) and specialization (changing random bits to 1's and 0's).

Effective rules could be evolved by the PPLS system. The learning system required a large number of rules, however, to provide the discovery algorithms with enough genetic raw material for the genetic algorithm to manipulate. Although the cover operators effectively made connections between the detector states (the stimulus) and the effector-generated actions (the response), the genetic algorithm provided the inference engine that manipulated this raw material until a useful set of rules evolved.

6.3 PDP-C

This section was to examine the PDP-C system interface and the performance of PPLS as an agent controller within it. The PDP-C part of the interface, however, did not correctly interpret the control information from PPLS (ignored it) due to a bug located in the PDP-C system code. Insufficient time remained to include any sustantial tests of that interface.

6.4 Discussion

The ability of the agent to support learning in on-line environments is a basic skill that allows the agent to adapt to unforeseen events in the environment. The tests have shown that a limited amount of learning can be supported while maintaining a reasonable agent behavior, though this is dependent on the rules created by the learning algorithm and varies from execution to execution.

Even if the agent "dies", however, the learning algorithm is given a useful piece of information that can be used to reweigh rules in an attempt to prevent the cause of death from recurring. This is a situation encountered in any concept learner that uses observation and experimentation to predict useful concepts (acceptable actions) within an environment(4). Eventually the learning system should amass enough concepts to adequately predict the best action in any situation encountered.

The PPLS learning model is simplistic, however, and this can be seen in the need to instruct the system (via a set of rewards) as to what is important in each phase of execution. If a set of operators were provided at the rule level to provide reward generation, then the phasing system built into the rule structure could also control the rewards associated with each detected environmental state.

One area of further study not addressed fully here is the implementation of a dietic detector state representation that changes based on the context of the situation (see Section 3.3.2). This is implemented to some degree in this system by using a generic distance field that measures the distance to whatever is the current object of interest: a target to approach or an enemy to evade. Such state representations can reduce the needed number of bits in the message fields and allow either a simpler detector space to search or a more complex environment to be encoded.

Further study of Booker's methods of niche generation seem appropriate to allow multiple concepts to be learned and stored (within the rule population) during each phase(4). One reason a large rule population was needed was to prevent convergence of the entire population on one concept. Niching techniques may be one alternative to reducing this problem.

Another area to be investigated is the creation of a more robust internal model of the outside world within the learning system. This model currently is represented by the rules (concepts) learned and used by the system. A *predictive model* that is updated based on the observed behaviors of objects in the environment can provide inputs into the state representation of the system. For instance, a field in the detector message structure may warn the learning system of the *anticipated* approach of enemy fighters when it enters enemy airspace. This piece of information can let the learning system react in a *predictive* way, thus implementing lookahead without rule chaining. This is similar to the two-level approach used by Booker to isolate environment prediction from action determination (see Section 3.4.2). This division of prediction from action agrees well with the distributed approach we propose.

The learning of phases might be implemented as follows: The initial phases of the system are set up as for the previous tests. Then, whenever the system encounters a situation it hasn't seen before (possibly measured by a lack of useful rules), the system creates a new phase and places the learning system in it. When the situation again changes (as measured by a significant change in detector state, for instance), a jump to a phase that has matching rules is made. A PHASE RETURN rule action could be used to return the learning system to the phase before the new phase, providing a subroutine approach to solving a problem.

A useful mechanism for the PPLS system would be one where *trap rules* could be implemented to prevent the system from performing actions that it just shouldn't do. For example, while the system was learning, it had the habit of dropping ordnances on any location it happened to be over. A strong penalty steered the learning system from this behavior, but it is disturbing nonetheless. The trap rules would ensure that certain conditions (such as being over a valid target) were met before the action could be executed. Such rules could be programmed into the system with the phase rules to guide the systems behavior and keep it within "respectable" norms.

Many potential improvements are possible. Some of these were presented earlier in this thesis, while others can be found in the classifier and machine learning literature (such as in (36) and the FTP archives noted in Section 7.8).

6.5 Summary

We have seen in this chapter how the PPLS system performs in both simple and more complex environments using implanted rules and rules generated by the discovery algorithms. By effectively limiting context (the domain space) the learning algorithm can be made to perform efficiently though, as in any trial and error learning system, the performance is significantly degraded when the system is learning new concepts to handle a new situation.

The CFSC-1 system provides a wealth of tools to address the learning problem. Some of these tools, however, were specialised for the test domains of their author(59) and need modification to be used in an agent control application. Also, the reused code is rather messy and undocumented in parts (especially the areas that implement the discovery algorithms) and this hindered tracing the actual reasons for many observed system behaviors. Further commenting and program execution tracing should be done if this code, the standard public domain classifier system, is used for further work.

Many new features can be added to the PPLS system to facilitate more complex learning strategies. These include the modified phasing and reward operators of the last section and more generic refinements of the system in general. Other methods, such as instructed learning, could be added to allow the system to learn new rules as it interacts with its environment. Finally, better monitoring facilities can be implemented to better follow why the system generates its responses, allowing improved insight into how the system can be further improved.

The next chapter pulls together the results of there tests and relates them to the topics discussed in previous chapters. We show that the system implemented has the potential to address

the criteria we presented in Chapter 1. We also address other issues concerning the implementing of autonomous aircraft agents in simulated environments. Finally, conclusions and areas of future research are presented.

VII. Conclusions

This chapter summarizes the results of this research effort, presents conclusions and suggests future areas of investigation. We first examine the efficiency of the PPLS learning system, what it was able to do, and how it might be improved. Though empirical in nature, these results suggest limitations in the implementation as well as guide later examination. We also evaluate how well the criteria and goals addressed in Chapter 1 (see Section 1.3) were achieved.

We then analyze the environment interfaces and how they both helped and hindered the learning system in achieving its goals. We critique each of the environments and their interfacing needs.

We then present an analysis of the scalability and practicality of this approach and discuss possible parallel implementations of PPLS. We also address the practicality of using this architecture in larger, more complex domains. All are issues that must be considered if an architecture is to be effectively used in complex simulation environments.

Finally, we present the conclusions of this successful investigation and note future research potential.

7.1 Summary of Results

This section summarizes the results of the testing described in Chapter 6. The specific test data are in an appendix (Volume II) of this thesis.

The implemented system showed an ability to control the agent's activities in the test environment. This demonstrated that a rule-based classifier system could provide an adequate control structure. Also demonstrated was the system's ability to interface with this environment and control the behavior of the agent via the effector interfaces. These are both important in an agent controlling task.

The system also demonstrated that a phasing system implemented within a standard classifier system could effectively control the focus of the system to a specific task and that this focus could be changed under the control of the detector interface and ultimately the environment. This capability is important when the agent's mission consists of a set of tasks to be accomplished. The phasing method is one way to implement a rule-niching scheme within the classifier system and results showed that it was effective in this capacity.

What was not adequately shown was the ability of the system to "quickly" learn rules in a new environment. I do not believe that this is a problem with the environment or the basic structure of the implemented system, but a result of the poor discovery operators originally encoded into the PPLS system. Better operators are currently being tested and support the conclusions that follow, but insufficient time was available to incorporate them here. These results are presented instead in an appendix.

Also presented in the appendix are later test results using the PDP-C interface that were not included in this thesis for lack of space and due to their late derivation. By making minor changes in the detector message encoding and incorporating new discovery operators, some interesting results were obtained. An analysis of these results are also included here. Most of the conclusions that follow, however, are not restricted to the specific test results.

7.2 *Meeting the Research Goals*

One question to ask is if the implemented system meets the criteria we set out to achieve. These criteria are repeated below (see Section 1.3):

1 The learning system is to demonstrate control of a simulated aircraft in each of the target environments.

2 The learning system is to perform a multiple-goal task to show that it is capable of controlling the aircraft and bringing it through an entire mission (sequence of tasks).

3 The controlled aircraft is to interact with its environment in a simple but "intelligent" way.

4 Real-time execution and interaction should be addressed.

a. The learning system is to demonstrate control of a simulated aircraft in each of the target environments.

The control mechanism uses a new phasing system that selected a subset of the rule base as the active rule set of each mission phase. By isolating each subset of rules, niches in the rule population could develop that specialised in handling the situations that occurred in that phase. The phasing system essentially added another level of control to the flat rule structure of standard classifier systems (see Section 5.3).

Effective control of the autonomous aircraft agent was demonstrated for tests using the internal test environment under simple conditions. Execution stepped between the different phases of the test mission and the rewards of each mission guided rule creation and credit allocation. Adaptation of the rule set under changing environment conditions was demonstrated (see Chapter 6).

Though simplistic, the rule structure should be scalable to larger rule sets and more complex detector and effector spaces. The literature suggests a limit on this scaling, however, based on the overall complexity of the problem (56). This can be mitigated to some degree by the use of heuristics in the discovery learning algorithms to guide the system in its search for rules to try.

The utility of this approach in other environments was not demonstrated due to implementation problems in the target PDP-C system in one case, and the early stage of development for the DIS interface and environment. I believe, however, that this approach can be applied to these other environments since the interfacing layers should be able to isolate the learning system from the environment differences. Although the rules may not port to the new environments, and the rules structure itself may have to be changed, the general approach and the implementation here

should be interfaceable to these environments with minimal actual software change (other than the data conversions of the interface layers).

b. The learning system is to perform a multiple-goal task to show that it is capable of controlling the aircraft and bringing it through an entire mission (sequence of tasks).

The phasing system performed well as a control sequencer system. With the discovery algorithms turned off, the system performed as expected, effectively activating sets of rules applicable to each task to perform. The ability of the system to jump between phases, as when the enemy came into view, was avoided and vanished from view, then came back into view and again had to be avoided, shows that control of the phasing system can adapt to the needs of the agent and condition in the environment. This is important since a purely sequencial rule phasing mechanism cannot adopt to such situations.

When the discovery algorithms were enabled, the system continued to perform the phasing function well. The discovery algorithms were not allowed to use the PHASE or STOP rules as parents for other rules and were not allowed to select these rules for replacement. This was necessary to ensure the stability of the control structure[1]. By limiting the discovery operations to changing the rules in the current phase, rule isolation could be maintained. This was important to the system "remembering" the rules of other phases, since the discovery algorithms tended to use the entire rule space to find new rules in untrained situations[2]. Rule isolation prevented this since it prevented changes to the other subpopulations. Each phase was also restricted to a fixed maximum percentage of the changing rule population to prevent a phase from adding new rules until it took over the existing free space[3].

[1] An area of future research noted later is providing the learning system a way to modify the phasing structure, thus allowing it to essentially write its own agenda.

[2] This is known as *convergence* in the genetic algorithm literature. See Appendix A.

[3] Since the rules of each phase are protected from replacement by another phase, however, this restriction may not be needed. By allowing rules of all phases with below average fitness to be replaced by any other phase, removing this restriction would allow a "pool" of replaceable rules to develop and be used as a scratch pad by the currently active phase. Further tests are needed to determine which approach is best.

The results of these learning tests showed that rule creation can be supported within the phasing structure and that rule isolation prevented new rule creation runaway, as expected. The efficiency of the system in developing new rules that were useful varied, however, depending on the discovery algorithms used and the seeding of the initial population available to a phase. One question that came up was whether to allow the discovery algorithms access to the rules of other phases as parents and so as seed rules. If the rules in such a parent phase are similar (such as between the Go to Target and the Go to Base phases), this enhanced the learning in the system. If the rules are counter or irrelevant, then this could have a disrupting effect on agent learning. For these tests the algorithms were given access to the entire population[4] If a *dietic* approach is used to message encoding, then an isolation approach would probably be needed, since the interpretation of detector messages would change as the phase changed.

c. The controlled aircraft is to interact with its environment in a simple but "intelligent" way.

The rules and interface structure implemented limits the learning environment and so how the learning system had to adapt. This was intentional to allow measured tests of the performance of the structure. For the environment tested, the results were very positive – the agent interacted with its environment in a productive way. This must be qualified, however, when the learning algorithms are at work, since any trial and error process is prone to produce negative results on occasion. (How to minimise this is addressed later.)

The question remains, however, as to how much learning was actually done. Using the measures addressed in Section 2.1 where we defined learning to be the improvement of performance within a specific environment, then successful mission completion in a minimum number of steps would be one such measure. Most of the tests showed that the learning system could perform better as the rules in the system evolved, and so showed that the system did indeed learn.

[4] This is controlled by a system variable.

The rate of learning was largely dependent on two characteristics: the discovery learning algorithms and the seed rules in the system. Better algorithms that used heuristics to specifically build potentially useful rules (such as rules that correctly specified an action the system could perform) greatly enhanced the learning process. The learning performance also greatly increased when seed rules that operated on the appropriate inputs were available to the genetic algorithm as models to build new rules from. In an entirely new situation, these seed rules might come from a randomly generated population of new rules, if the population was large enough to produce useful seeds.

Though much improvement is possible in the system, it did perform in an "acceptable" fashion most of the time. There is still some tendency for brittleness to form in the system as specific examples are used to train the system. The input data representation and the set of training examples used can go a long way at minimizing this brittleness in these systems.

d. Real-time execution and interaction should be addressed.

The execution rate of the core learning system (minus file input and output) is quite fast. The complexity was noted as roughly $O(Nm^2\ Nc)$, where Nm is the number of messages (on average) each cycle and Nc the number of rules in the system At present the interfaces used are all I/O bound, so measurements of the system's speed while interfaced to an environment is not possible. Execution of the system with a loaded rule base and no interfaces but a simple PPLS test environment built into the system show execution rates of 100 to 1000 cycles per second on a Sun Sparc2 workstation. This is encouraging, since the system has the potential to be parallel distributed to further increase this execution time.

Even so, execution rates of many times a second (generally governed by the speed of screen I/O) were common for the previous tests. As long as multiple-cycle reasoning is not implemented (such as look-ahead) the system should meet most simulation real time requirements. Even with

these additions, the overall execution time is still within a 0.1 sec update rate that is usually sufficient to control aircraft agents(8).

7.3 Enhancing Learning

One way to enhance learning in the PPLS system is to limit input and output complexity through the use of filtering. As Greffenstette noted (Section 3.4.2), the tendency of these systems to fail to find useful concepts (rules) increases quickly as the domain space is increased. This is the reason we have argued for keeping the detector and action spaces of the learning algorithm simple and leave the "grunt work" to the filter interfaces. Any deductive or formula transformations in particular are inappropriate to the concept learning system. These should be performed by other subagents of the autonomous agent and the simplified results mapped to the learning system's input and output spaces.

Second, efficiency must be based on the needs of the application. 99.9 % efficiency in learning concepts from examples may be needed in some environments, but a much less efficient system may be useful in other environments. The tolerance the environment has for mistakes is a key factor. Another is the types of mistakes the agent can make. Flying in the wrong direction may not be detrimental to the agent, but flying into another aircraft might be. The measure of efficiency in learning concepts has to take into account the utility of the learned concepts. The idea of restricting the actions of the agent to those that "make sense" is another form of filtering. When the learning system tries to do something that is easily detected as not desirable, the interface should refuse to do it and indicate the error to the learning system. An example is when the learning system tries to drop an ordnance while not over a target.

One might argue that we are telling the system everything and leaving nothing to the learning algorithm. This is not true. We may be providing the criteria and gauges of performance, but not the actual execution of actions to get to it. The system is being told to work at meeting some goal

or objective, and then being left to develop the rules to reach the goal. If the goal is possible, and the rewards sufficient to guide the system, the actions necessary for each detected situation of the environment will be learned.

What feedback knowledge is supplied to the agent is also important. The learning system must be able to measure the success (via rewards) the effectiveness of the changes it has made in the environment (perceived via the detector interface). This cause and effect relationship allows the system to build a model of its environment via the rules in its rule base. In complex systems it may be that only experience over time will provide sufficient feedback to the agent as to which concepts are actually useful. Therefore it may be difficult to judge the worth of a concept taken outside of a useful context.

The forming of relatively isolated population niches also seems important to the efficiency of a classifier-based learning system. These systems have a tendency to generate volumes of rules addressing the current problem, which is one reason they are so efficient(36). This action also forces other useful rules out of a limited size population, which leads the learning system to "forget" concepts it has just learned. The phasing approach we presented is one attempt to limit this population growth. Others, such as Booker's restricted mating policies and environmental prediction-action separation[5], are others (see Section 3.4.2). To handle more complex environments, implementing subpopulations within phases to allow population niches to service the specific needs of the agent may be essential.

The learning system architecture of PPLS has the potential to be an efficient and useful decision maker in a complex simulation agent. The environmental filters and the concept learning system should be taken as interdependent parts of a learning system architecture.

[5]I.e. the separating of environmental prediction rules from action generation rules.

7.4 The Environment Interfaces

The interfaces designed for the PPLS system have attempted to standardise the form of information at each side of the interfaces. This format of the data structures is determined on the environment side by the environment being connected to the system. On the PPLS side, we have tried to maintain a consistent data structure and mapping. By treating the interface itself as a layer in the interface structure or a filter that converts information on one side of this layer to the form needed by the other, any similar environment should be interfaceable to the PPLS system if an appropriate interface filter is added to the interface.

The rest of this section discusses how the environment influenced the form of each interface and how this effects learning in the system. Emphasis is on mapping the actual ***decision domain*** (the domain the agent must make decisions about) from the rest of the environment inputs and outputs.

The Internal Environment

The internal test environment was based on the PDP-C interface and so is file oriented. The simulation provided by this interface was crude but provided a sufficiently robust interaction to allow the system to learn useful behaviors. Since no other interface was reliably operational by the end of this research effort, no tests could be made as to the portability of the rules learned in this environment.

By reducing the variables (state dimensions) of this environment, the learning process was made easier. However, many of the complex relationships of a more sustantial aircraft environment should be mappable to this simple set of detector inputs. And adding additional output functions would not be that difficult either. This allows the overall PPLS learning system component to be mapped to more complex domains by changing the message format (to add more fields and change others, as necessary) and the input and output filters.

PDP-C Interface

In any simulation that is linked to real time, the synchronisation of the simulation to the learning system is desired[6]. This synchronisation was achieved via a set of data files and a set of semaphore files. The use of Unix pipes was used to force each program to stop until the data written by one was read by another.

This approach was workable, but depended on setting up Unix named piped for the communications paths. It also limited the control of the data paths to some degree[7] A more versatile approach would be to use Unix socket connections to provide the communications. This would allow for better monitoring of the status of the pipes (socket pairs) thus generated, as well as more control over the status of the l' .es. However, the ability to do this in Clips is not clear at this point.

The need to read and write COOL object records forced the development of the `slot_io` package for the PPLS system. Th s package is still not what it should be, since it interprets the object records line by line, and t e text object records use parentheses to delimit record slots. A better package would pars the parentheses and build the internal slot representations based on this parsing. This car get very complex, however, especially since the slots have to be written back out in a COOL-read ble form as after changes to the slots are made. There was no t me to pursue this, however.

Finally, as wit any such interface, the interpretation of the data quantities bec me a problem. The PPLS syste uses a degree-based, compass-like orientatio system, for instance, while PDP-C uses a radians-based trigonometric-like system. The interface layer provided the conversions, but this added ano her area of uncertainty into the interface that had to be tests. But this is what interfaces are or – to isolate these dependencies so that other parts of the system don't have to deal with them. The interface package did this well.

[6] Alternatively, both systems could run independently and synchronise via event coordination or time stamps (as in DIS).

[7] This was more a limitation of the Clips/COOL language than of the named pipes.

Limitations of the PDP-C Environment

The PDP-C system currently runs at about one update per second. This update rate is rather slow for interacting with human-controlled objects in a DIS environment. And even this rate is not guaranteed. A key problem with the PDP-C implementation is its heavy dependence on external files. Many delays in execution I believe can be traced to the poor performance of the AFIT ethernet and file servers, which handle over a hundred workstation and other systems continuously. If all file I/O was replaced by named pipes to waiting processes on the same workstation (such as logging processes, etc.), this might speed things up considerably[8]. Running the system on a multiprocessor Silicon Graphics system, with the agents distributed, may also help. Other methods are also possible, but are more involved than these.

One other limitation of the PDP-C system is the complexity increase each new agent adds to the system. As noted earlier, a DIS interface would likely require each external object to be internally simulated as a craft agent, probably as another edcraft. In a complex world there might be many things interacting with the PDP-C simulation at once, which might overload the system rather quickly. Also, the system does not support many external world features (such as mountains and tanks), and so would be limited to air engagements without some modifications to its internal world simulation.

Interfacing to DIS

Once the PDP-C interfacing requirements are removed, the PPLS system has the potential for much faster operation. The minus side is that the system would require agents to take care of all the functions now being done by the PDP-C simulation environment. These include radar detection, flight equation implementation tracking of other simulation objects, etc. Much of this code currently exists, however, and a complete system should be buildable.

[8]No actual measurements were made, but other tests suggest a factor of ten speedup may be possible. What is not known is how much of the processing time was spent executing input and output and how much was taken by the rule matching and execution operations. The system has been observed to slow down considerably during some phases of its operation while maintaining the same levels of file access.

The Distributive Interactive Simulation interface proposed should let the PPLS system be interfaced to an active DIS simulation net. There was no time to test this, however, and the DIS interfacing efforts continue. By adding the needed layers of filtering, including a realistic flight model, the system should be able to interact with its environment. The message encoding would need change, however, since the number of variables would increase, and each would need to be represented (in as simple a manner as possible) as a field in the detector messages. Also, the reward system would need to be modified if multi-agent interactions are needed. (Only one enemy is supported at this time, and no friendly aircraft are represented in the detector state model implemented.) These changes should allow for a simplistic agent behavior that can be used as a starting point for more complex agents.

The DIS interfacing code is currently being written by another AFIT researcher and should be ready early next year. An object-oriented approach should make the inclusion of multiple simulation agents based on the DIS PDU message system relatively simple. If dead reckoning[9] is ignored in the system (i.e. objects are taken to be at their reported locations and are static until the next update), then a set of objects that merely store the information provided in the received PDUs would create a simple system that could later be updated to include more sophisticated techniques. If designed right, such a DIS interface could be used with other systems besides PPLS. The key is in the layered interface as detailed in Section 4.3.

7.5 Practicality and Scalability of PPLS

This section looks at the practicality of scaling the PPLS system to more realistic levels of domain complexity. Such a scaling may involve increasing either the domain space, the action

[9] *Dead reckoning* is a technique used in the DIS standard to implement an optimistic discrete event simulation environment. Since updates on slow networks are spaced relatively far apart in time, each participating simulation uses dead reckoning techniques to determine where the object would be given its last reported location, velocity, and acceleration - pretty much as if the object was "dead" and just coasting. The problem is that the guess predicted by the observing simulation might be wrong, in which case past events based on this information must be "fixed". This has interesting effects when someone who was "shot down" suddenly isn't and is now shooting at you(19:74)...

space, or both and may require additional filtering in the interfaces. Additional systems may also be interfaced to the learning system, requiring coordination of these inputs. The actions required of the agent may be more complex and require sequences of actions to be learned. Finally the message list, rule list, and other system storage may need to be increased to handle the larger number of possible concepts the system must handle. These concerns are addressed in this section.

7.5.1 Limiting complexity. As noted previously in this chapter, genetics-based classifier systems (and concept learning systems on the whole) must search exponentially larger rule spaces as the number of possible input states and the number of possible actions increases[10]. This limits practically the sizes of the input and output spaces a system can deal with to a relatively small value. Though genetics-based classifiers can process large rule bases dealing with large detector and effector spaces, the complexity of the rule space limits the effectiveness of the discovery algorithms and therefore the ability of these systems to adapt to their environment. This is one of the limitations that must be overcome in a practical learning system that can deal with a complex environment.

As noted throughout this thesis, one such approach to limiting the rule space complexity is to limit the possible number of states and actions the system must recognize and generate. Since the complexity of the environment can't typically be decreased, we propose using a set of transformations to map the significant characteristics of the environment to the learning system domain and another set of transformations to map the outputs of the learning system back to the environment. We called these transformations filters and interface layers previously, but the function is the same: to take the complex characteristics of the environment and represent them in a simpler way that is within the limits of the learning system to deal with.

This approach implies that such a mapping is possible and that the complexity of the process is within the processing constraints of the real-time agent system. We argue that most of the filtering needs of the system are infact typically deterministic processes. For example, the derivation of

[10] Assuming an unfiltered interface to the domain.

the Target Left signal to the learning system is derived through a set of calculations based on orientations and offset angles. These angles could have been encoded into the learning system detector messages, but would have added considerable complexity to the detector space. The required information (in this simple system) was just a relative measure. Even in more complex environments, most of the relevant information can be represented in a simple form of low cardinality without significant loss of discriminating potential. We believe that this is where most of the filtering in autonomous agents must take place[11].

Complex actions can also be represented simply to a large extent. The autonomous aircraft agent can learn that a maximum turn rate dive is the best maneuver in a particular situation. How the maneuver is actually done need not be known to the learning system for it to decide that it is the right maneuver for the situation. The maneuver can be passed to an interface filter that carries out the maneuver itself in an autonomous way. The key to this approach, we argue, is to provide enough subagents to accomplish the generation of any behaviors needed by the agent within the time and processing limitations.

Thus we treat the inputs to the overall agent as inputs to sensors that preprocess information to the learning system and the outputs as the responses of subagents to control signals from the learning system. This approach isolates the learning system within a filtered model of the environment that can be simple enough to apply discovery learning techniques. If the layering and filtering of data is sufficient (more than one layer can exist between the environment and the learning system, and, in fact, different inputs and outputs be filtered differently) then simple behaviors can effectively control complex environmental actions and reactions.

[11] To further make this point, animals, as the ethologists tell us, are driven by relatively basic and simple decision processes. These processes rely heavily on the animals' senses and other preprocessing subsystems to filter the rays of light entering their eyes, for instance to the simple relationship that, say, a bone is on the dinner table. A dog wanting the bone probably does not perform any specific calculations, but instead reacts to the relative positions of the objects and a set of simple rules that describe them. The dog leans its forepaws on the edge of the table (which it believes will support it), reaches with its mouth until it makes contact, and grabs the bone. In a similar way the complex data from the environment can usually be filtered to a simple level within the learning ability of a simple discovery learning system.

Another way to reduce the input detector space complexity is to use a dietic approach where the detector space is dependent on the context of the situation. Dietic representations allow the input space to be time-sequenced in that the only inputs provided the learning system are those needed to make a decision about the current situation. This is quite similar to our use of phases to limit the effects of a rule subpopulation to those situations for which the rules apply. Dietic representations just expand on this by removing the input information not used by a rule subpopulation when that subpopulation is active. An example based on the PPLS system is to remove the base location information from the detector messages when the system is in Phase 0 and trying to locate and destroy a target. This information is not needed during this phase and merely adds length to the message (and increases the state space). Such a method, though, requires controls to ensure that the detector space is properly interpreted[12]. Enforcing phase isolation is one method of doing this.

Further filtering is possible in such agent systems by placing constraints on the allowable actions of the system. The PPLS system, for example, does not use a complete action encoding, and so there are some action codes that are invalid. New rules created by the discovery algorithms should not use these codes since they cannot lead to effective actions (except doing nothing). Some actions are also invalid in certain contexts. For instance, the dropping ordnance action is inappropriate anywhere except above a valid target. This can be checked in the effector interface and the learning system scolded (penalized) whenever it tries to drop an ordnance on the wrong place. Other checks can be done heuristically to limit the actions in rules to those that have potential to be useful[13].

These methods can limit complexity of the domain and action spaces to a point that the genetics-based classifier system can be an effective rule discovery and processing system and a

[12]Otherwise a rule expecting to find the target location in field 4, for instance, might find its own base position and go off and destroy that instead.

[13]The actual usefulness of a rule can only be determined by the agent through experience in the environment. A perfectly "good" heuristic can generate rules that are just not effective in a particular environmental state.

viable controller for autonomous aircraft agents. There is always a penalty for such filtering, since it prevents the learning system from reacting to the "raw" inputs from the environment – it becomes dependent on the interpretations passed it from the filters. It is for this reason most that we argue in favor of keeping the complexity of all input and output filters simple to allow them to be reliably implemented and tested. complex filters can lead to flaws in environmental interpretation and so to strange or even drastically wrong behaviors[14].

7.5.2 Is concept learning appropriate? This is a question that needs to be asked. One of the major premises of this research is that all situations not requiring an internal chaining of states can be handled by the concept learning approach. Since the condition parts of concept (decision) rules can represent nearly anything, they can encode the current state of the environment, the available actions, and the expected outcomes of these actions. The learner then could decide if a particular course of action is fruitful based on its ability to label each such state as useful (i.e. apply the concept of "useful" to the input examples) and then choose the most action decided to be the (potentially) most useful. We saw that such a system would be essentially a *reactive system*, but still could handle complex time-dependent behaviors as long as the environmental state presented it included sufficient time-dependent data. In this way concept learning can be applied to any situation that the agent may find itself in, if the filter interfaces provide the right data.

Multiple decision possibilities are then represented by multiple rules that each represent a recommended action for a given state. The system then chooses these actions by choosing the rule to fire based on its fitness which is based on the rules past performance. The rules of each niche in the rule base thus address how a particular situation can be modeled as a set of concepts. Each concept is an action the system can perform and the classification being done is to place the system state represented by the current detector messages into the appropriate action concept.

[14] If the interpretation is consistent and the rewards appropriate, however, then the system might still learn to react appropriately, in effect learning to correctly map the wrong data to correct reactions.

7.5.3 Multiple sources of inputs. Multiple input sources should not be a problem since they can be encoded into the single message vector. The PPLS architecture allows, in fact, multiple detector messages to be generated by the interface. If each message is somehow tagged as to source, then complex encodings (of a dietic form) are possible.

If the multiple inputs can be time-divided, however, then a phased approach would allow each such input source to be handled in relative isolation, and so keep the complexity of the domain space down. This is important if effective learning is to be maintained. The interface layers should handle the time-sharing of the learning system, as discussed in earlier chapters.

7.5.4 Single-goal hypothesis. One limitation built into the implemented system and the phased approach is the ***single goal hypothesis***, which we define as the limiting of the learning system to obtaining a single goal at a time. This was necessary to allow the activation of a set of rules appropriate to a specific task under control of the phasing mechanism. In this section we analyze the effects of this limitation.

First we note that this limiting assumption is not unique to PPLS. Many systems (such as the MAXIM and PDP-C systems already discussed) use an agenda to focus the system on a specific task at a time. Though multiple objectives may be coded via rote learning into the system, this agenda approach is intended to specifically limit the system and allow a clear logical flow from one specific goal to the next.

Modifying the agenda allows some variation in action, such as when a new environmental state requires a change in the agenda of an agent. Still, although the agenda can be said to adapt to changing conditions, only one goal at a time is being pursued by the system. Otherwise a form of conflict resolution would be necessary when reaching the goals require conflicting actions.

The implemented learning system PPLS, using a stochastic parallel rule firing approach is possibly less bound by this assumption than many other approaches. Since the actual rule fired at any time is based on the expected utility of the rule given an input state, the system is free to

switch between goals every other cycle if the rules for the two goals are in the same subpopulation and the rewards from the environment support this activity. Though each phase is thought of as isolating the rules for a particular mission task, in effect all the phasing mechanism really does is set up subpopulations meeting the current needs of the environment in that given environmental state. Thus the PPLS system is capable of learning (via implantation or via discovery) to deal with multiple goals. We make the single-goal hypothesis in this system only to organise our human view of the problem. In a complex environment it may be likely that the system learns a complex set of rules that throws this hypothesis out and so is hard to interpret[15], but works fine in the given environment just the same.

7.5.5 Comparison to other approaches. This approach is comparable to other approaches being used at AFIT to control autonomous agents. It is a rule driven system with an agenda (implemented via the rule strengths and also by the detector interface as the states presented to the system trigger a sequence of rule niches) that is overall governed by a mission-executing phasing mechanism. Without the Bucket Brigade to update rule strengths or the discovery algorithms to change the rules, the system would be very similar to other rule processing systems, except for the trinary alphabet low-level rule encodings.

Adding the bucket brigade to predict the utility of rules (credit allocation) and the discovery algorithms to modify the knowledge structure (discovery learning) provides capabilities in PPLS not available in many other approaches. PPLS works (for now) at a relatively primitive level of detector and action encodings. However, the system implements to some degree all the learning strategies we believe important to autonomous agents (including rote, deductive, and inductive) and no system feature expressly limits building on this implemented frame work towards systems that can handle more complex environments. It is this pulling together of the various learning strategies,

[15] This is related to the idea of *subsymbolic learning* that classifiers and neural networks can exhibit(48).

as well as symbolic and subsymbolic representations, that make these systems so potentially useful in controlling autonomous agents.

7.5.6 Environment interaction. Can the PPLS system handle the complications of a real environment? I believe simplicity is the key here. Each part of the system performs a relatively simple function, including the learning system. This means that additional (possibly learning) subagents are added to the system as the tasks of the system increase to maintain the complexity levels to manageable levels at all points in the PPLS system. Also important is the use of all the types of learning described in Chapter 2, including the implantation of "canned" rules into the systems to get them started. Building a pilot is not an easy task, and we believe that it should not be tried for in one super-complex system. A distributed system of simple parts is (potentially) much easier to track and maintain.

7.6 Parallel Implementation Potential of PPLS

The system as it stands has many components that can be distributed over a parallel architecture or a network system. The two areas to examine are the filter interfaces and the core CFSC system.

The filter interfaces are modeled as objects with fixed input and output data structures and should be easy to distribute between nodes. The key here is to minimize the data exchange between these filters to reduce coupling and communications costs.

The complexity of the CFSC system is mostly based on the processing of each message by each rule via the matching process. This matching process is mainly independent for each rule, so one potential partition is along rule boundaries. Robertson has shown that the CFSC system can be implemented on a CM-5 system with one rule on each node(59). Such a partitioning would not be too difficult for the current system. Messages would still need to be distributed, adding communications costs, but the computational complexity of the system would be now $O(Nm^2)$,

with the number of messages (Nm) being much smaller typically than the number of rules in a large system (Nc). This could be a large gain over the current $O(Nc * Nm^2)$ of the system. Further parallelisation may be possible, but would increase the coupling between nodes. The effects of this need to be examined before this is done.

The other features of PPLS should not severely restrict a parallel implementation, since most of the complexity of the core loops are individual checks that can be parallelisable. Dividing the rule base up and placing it on different nodes seems the best way to do this. The message list provides most of the system interaction and would best be accessible by each node, forcing some form of update of each node's copy as required or the use of shared memory. Otherwise, there should be no limit to how finely the rule base is divided. This gives real speed potential to the system as a whole.

7.7 Conclusions

The overall results of this investigation show that genetics-based classifier systems can be used to implement agent control learning systems within certain contexts. This section summarizes the points presented earlier in this chapter and in this thesis.

The general approach used in this investigation was to break larger problems into smaller ones. This approach led to the biased rule niching approach and to the distributed interface filtering architecture. Both of these limit the complexity the learning system must deal with.

This approach was demonstrated in Chapter 6 by the implemented Phased Pilot Learning System (PPLS) and was able to navigate an agent through a simple environment where the effectiveness of a behavior of the agent depended on the state the system was in. There was no significant limitation that prevented the enlargement of the phases of the system to include more recognized mission tasks and the overall approach allows the system to shift rule subpopulations as

these phases are recognised via the detector messages and the conditions of the rules that govern phasing.

Though not tested, the potential for phase creation during a mission was noted and would allow the agent to create a new set of rules, separate from the successful rules of other phases, to address the requirements of a new untrained situation. The system as its implemented allows the rules in other phases to act as seed rules, which increase phase learnign efficiency if the rules are similar. Otherwise, the discovery algorithms generate new rules that are evolved by the genetic algorithm into a subpopulation that meets the need of the new niche.

The interface filtering allowed the learning system to concentrate on the relevant aspects of the environment, filtering out details not important to the current decision process. Though this filtering approach places a burden on the filters to provide useful and relevant data, this approach was shown to minimize rule space complexity and allow the system to adapt effectively. How much filtering to provide is a trade-off issue and is dependent on what defines a state in the environment. Enough information must be presented to the system to allow the concepts that divide the states into appropriate actions to form.

The filtering approach also matches other models, such as the DIS architecture model, and promotes portability of the system to different environments. Though other environments were not tested, the design for these interfaces (especially the PDP-C system interface) was shown to be easy enough to implement. More complex environments would require more filtering, but keeping each layer simple should make such an implementation easier to build and test.

The PPLS system implements various learning strategies, including rote learning (via rule implantation), deduction (via filters that process input data to usable forms and filters that translate the responses of the system to action sequences valid in the environment), and induction (via the discovery learning algorithms). Each of these learning strategies play an important role in allowing the agent to effectively perform in a changing environment with a minimum of rules.

Overall, the design was shown to meet the objectives stated in Chapter 1. The system implements various learning strategies and uses distributed filter interfacing to reduce agent control to manageable levels. The phasing rule system allows concepts to be learned by the system and remembered for future use. Each of these components are important to allowing the PPLS system to effectively control an agent. The implemented system showed how it is feasible to implement this design and get it to work. The algorithms of the system are still rather crude, however, and further refinements should provide better performance still.

7.8 Future Research

Many areas of potential future research exist. The creation of better discovery operators, possibly using environment-dependent heuristics, would greatly enhance the rule discovery process. The current operators, mainly modified versions of the "canned" CFSC mechanisms, are still quite inefficient. Better discovery operators and triggering mechanisms for these operators are needed.

What should be filtered and what shouldn't needs to be carefully addressed. This issue has implications to all agent controllers, not just PPLS, and can form the basis for a standardized network interfacing system (perhaps based on the DIS model) that other autonomous agent systems could benefit from.

Better encoding of information into the message structure is possible. Many current research studies[16] are addressing the different ways that niching can be used to enhance performance in classifier systems and how best to encode concept information. These methods are directly applicable to the PPLS system and should be easy enough to implement in the structured PPLS and CFSC source.

In summary, much is left to do. This research effort should be considered just what it is: a feasibility study that shows the potential for further investgations.

[16] For instance, the latest papers from the Navy's Artificial Intelligence Center (AIC), which are available via anonymous FTP to FTP.AIC.NRL.NAVY.MIL that there was no time to include here.

Appendix A. A Review of Genetic Algorithms and Genetics-Based Learning

This Appendix reviews the basics of ***genetic algorithms*** and how they can be applied to machine learning applications.

A.1 Genetic Algorithms

Genetic algorithms are search algorithms based on adaptation and natural selection. They can be used to guide the construction and restructuring of knowledge representations within a system and so adapt the system to its environment. These systems use a building block representation where all possible representations of knowledge (such as the rules in a rule-based system, for instance) can be formed by assembling these building blocks in different ways. This allows a genetic algorithm to construct new knowledge representations out of existing ones using genetics-like operators (such as genetic crossover and mutation) that operate by rearranging these building blocks. Representations (population members) that perform better are more likely to be selected to propagate their characteristics (building blocks) to later offspring via the genetic algorithm (12, 22).

Genetic algorithms maintain a ***population*** of potential solutions from which they build new populations of revised potential solutions. For computational reasons, and to promote ***selective pressure*** on the candidate solutions, these populations are limited to some fixed size. The algorithm selects a percentage of these member solutions, performs a set of genetic-like operations on these parent solutions (*breeds* them), and replaces a percentage of the population with the new potential solutions.

Key to this process is a ***fitness function*** which is used to rate each solution's ability to solve the problem. Those solutions with higher fitness (the better solutions) are more likely to be selected for breeding, while those solutions with relatively low fitness (poor solutions) are most likely to be replaced by the children solutions generated during reproduction. Thus, due to the selective pressure the competition forms, a set of good solutions built on the good parts of other solutions

quickly forms. After a set number of breeding cycles, the most fit of the population members are taken as likely solution(s) to the problem being solved (22). Note that genetic algorithms are probabilistic-based, and can give different solutions when run multiple times on the same problem. For this reason, most applications that use genetic algorithms run the algorithm multiple times and take the best solutions of the runs. They can also miss the mark entirely and fail to converge, giving rise to the concept of *GA hard* problems that are difficult for GAs to solve.

Genetic algorithms are limited in their usefulness by the possible ways a population member can be encoded. The most typical method of encoding, that of a binary string, allows the algorithm to easily form new solutions by swapping bit strings between pairs of strings (i.e. the *crossover* operation). Other representations are possible, including integers, floating point numbers, and other symbolic forms. In the case of numbers, some arithmetic operation, such as incremental adjustment or averaging of parent values is used to derive a new value for a particular *allele* (gene position) in a child. In symbolic representations, some form of subsymbol modification is used, which makes the implementation of the genetic operators quite dependent on the solution representation used. Proper choice of the gene encoding method is crucial to a successful genetic search (10). This is especially true in machine learning applications, where many different methods have resulted in varying degrees of success (12).

A.2 A First Illustrating Example

The following example may make this a little more clear. Here, the task is to find the root of $y = x^2 - 25$. Now forget for the moment that one can just plug this into the quadratic formula and get $+5$ and -5. For simplicity, the range of x is also limited to the range $-8..7$, which is the binary representation of a 4-bit two's-complement integer. The problem, then, is to find those 4-bit two's-complement binary strings that make y zero.

The next step is to select a fitness function. This function is key to the genetic algorithm in that it is the only feedback available to it on how well the search is going. For this problem, f is chosen as $f(x) = 100 - abs(x^2 - 25)$, which results in a maximum of 100 when $x^2 - 25$ equals zero. Decreasing fitness values (i.e. best is lowest) can also be used, but is less intuitive. Note that the number 100 is not particularly special, but is just a number that keeps the result positive. (The range of f is $100 - (8^2 - 25) = +61$ to $100 - (5^2 - 25) = +100$.) The goal is now to find the 4-bit binary string encoding of x that maximises the fitness function.

The genetic algorithm requires a set of operators to manipulate these bit-string representations. The three most common are ***selection***, ***crossover***, and ***mutation***. These are described below.

Selection is how the algorithm choses which strings to operate on and provides the selective pressure that allows it to converge on to high fitness solutions. Typically those population members (strings) with highest fitness are most likely to get selected. Many ways exist to do this, but for this example the strings are sorted by decreasing fitness value and a form called ***roulette wheel selection*** is used where the fitnesses are added and a random number between zero and this sum is selected. The population members' fitnesses are then summed until the sum exceeds this random fraction of total fitness. Another selection scheme uses the fitnesses to rank the population and then using the rank to determine the member's chance of selection. This ***ranked-based*** scheme has showed promise in keeping a population from *prematurely converging*, which is what happens when a few high-fitness members of the population begin to receive exponentially-increasing numbers of offspring and so crowd out other, but potentially useful population members. The goal is to prevent coonvergence until an optimal (or at least sufficiently optimal) solution (member) is found. Until then convergence is to be avoided.

Crossover is used to exchange genetic information between two potential solutions in the hope of generating a better solution. It serves as the means to redistribute the higher fitness building

blocks of a population as it forms new candidate solutions from the higher fitness current members of the population.

Complexity.[1] The genetic algorithm is of polynomial order complexity with a finite space requirement determined by the population size. Although Goldberg (23) suggests that there is an optimal population size which depends upon the length of the string, there is no fixed dependence between the length of the string and the population size. Experimental evidence, however, suggests that an insufficient population size may adversely affect solution quality (22)(46). The terms in the order of the genetic algorithm reflect the length of the string as well as the number of strings in the population. An entire cycle of the genetic algorithm is executed up to a maximum number of generations specified by the user. The basic pseudo code for a genetic algorithm is presented below.

```
initialize population
calculate fitness for all members of the population
for i = 1 to max_number_of_generations (m)
  for j = 1 to population_size (n)
    crossover
    evaluate fitness
    mutation
  end loop
  selection
end loop
```

The various genetic operators each have associated maximum complexities although complexity of an actual implementation may differ. The crossover operator selects two strings from the population pool (n), picks a random location along the length of the string, and then swaps the two tails of the parent strings which follow the randomly selected crossover point. This portion can be considered to be $O(l)$ since it needs to traverse the length of the string.

The fitness function includes a call to decode the string representation into the value in the problem domain. This decode call could be an $O(1)$ or an $O(l)$ function, depending upon the string representation scheme and the programming environment capabilities. Evaluating the

[1] The following material is borrowed, with permission, from Olsan's work(53) and is a collaboration between him and Don Brinkmann.

fitness function should be an $O(1)$ operation since it simply substitutes the decoded string values, and evaluates the objective function. However, for complex problems, the evaluation of the fitness has a lower bound of the or der of the objective function —the functions being optimised.

Mutation also could be an $O(1)$ or an $O(n)$ operation depending upon the particular implementation and also the mutation strategy being used. Studies have shown good results are obtained with a mutation rate of once for every thousand string position t ransfers (22).

The selection function has an $O(n)$ complexity since it must implicitly or explicitly evaluate each member of the population to determine which strings will be carried on to the next generation. Actual implementations of the selection operator may be of $O(n^2)$ complexity, such as a roulette wheel approach biased according to the fitness of the strings.

This makes the complexity of the entire algorithm

$$O(m * max(n * max(l, \text{fitness function}, n), n^2))$$

which is equal to

$$O(m * n * max(l, \text{fitness function}, n)).$$

A.3 Theory

This section introduces schema notation of genetic algorithms. Then the fundamental theory shows that genetic algorithms produce increasingly better populations. This material is extracted from Merkle's thesis (46:18-20).

Schema. Goldberg develops an estimate for the performance of the SGA(22:28-33). Theoretical analysis of GA performance makes extensive use of *schemata*, or similarity templates. Schemata are strings composed of characters taken from the genetic alphabet, with the addition of the "don't care" character. A schema thereby describes a subset of the potential solutions. For example, the schema 1******* represents the set of all 8-bit strings which contain a 1 in the first position.

Likewise, the schema 1******0 represents the set of all 8-bit strings which begin with a 1 and end with a 0.

The defining length, $\delta(H)$, of a schema is the "distance" between the index of the first specified position and the index of the last specified position. For example, $\delta(1*****0*) = 7 - 1 = 6$, while $\delta(1*******) = 1 - 1 = 0$. The *order* of a schema H, which is denoted $o(H)$, is the number of specified positions in the schema. For example, $o(1*******) = 1$, while $o(11111111) = 8$.

The schema concept can be extended to apply to absolute and relative ordering problems. Following Kargupta(38), an absolute ordering schema defines a set of valid permutation strings. For example, the absolute o-schema ! 1 ! 5 ! ! represents the set of all permutation strings for which the second and fourth positions contain alleles 1 and 5, respectively. This o-schema is distinct from the standard schemata * 1 * 5 * * in that the former requires that the string represent a valid permutation, while the latter does not.

Following Goldberg(22), Kargupta uses $rs^l(H)$ to denote the set of all valid permutation strings in which the alleles specified in H occur in the specified order. For example, $rs^6(1,5)$ represents all permutation strings of length 6 in which the allele 5 occurs after the allele 1.

Fundamental Theorem. Defining the average fitness of a string matching a schema H to be $f(H)$, the average population fitness to be $\bar{f}$, and the number of strings in a population at time t which match the schema to be $m(H,t)$, the effect of the reproduction operator is

$$m(H,t+1) = m(H,t)\frac{f(H)}{\bar{f}} \tag{A.1}$$

Noting that crossover disrupts a schema only when the crossover point occurs within the defining length of the schema, the probability of survival under crossover for a schema in a string of length l is

$$p_s \geq 1 - p_c\frac{\delta(H)}{l-1} \tag{A.2}$$

where p_c is the probability of crossover and the inequality is used to reflect the fact that crossover may not actually disrupt the schema even when the crossover point is within the defining length.

The probability of survival for the above schema under the mutation operator then can be estimated as

$$p_{ms} \approx 1 - o(H)p_m, p_m \ll 1 \qquad (A.3)$$

where p_m is the probability of mutation. Combining these results and omitting negligible terms gives an estimate for the expected number of examples of a schema in the next generation:

$$m(H, t+1) \geq m(H, t)\frac{f(H)}{\bar{f}}\left[1 - p_c\frac{\delta(H)}{l-1} - o(H)p_m\right] \qquad (A.4)$$

This is referred to as the Fundamental Theorem of Genetic Algorithms, and can be interpreted as stating that "short, low-order, above-average schemata receive exponentially increasing trials in subsequent generations" (22:33). This result also goes by the name of the Schema Theorem.

Genetic algorithms have many advantages over conventional solution search techniques. First, they allow a natural evolution of solutions using a Darwinian survival-of-the-fittest approach to improve the knowledge in the system. Another advantage of genetics-based systems, which use populations of knowledge representations, is the inherent parallel nature of such systems, and the quick processing this allows (10, 22).

A.4 Applications

A genetic algorithm provides solutions to search problems. Since GAs require no knowledge of the problem, they are well suited for problems for which no known algorithm exists. Genetic algorithms may also be beneficial to problems in which all known algorithms take unacceptable time to execute.

This section divides search problems into two major categories: functional optimisation and combinatorial optimisation. An example of a functional optimisation is finding the minimum of $f(x, y) = 100*(x^2-y)^2+(1-x)^2$. An example of a combinatorial optimisation problem is finding the shortest circuit which contains all vertices in a fully connected graph (traveling salesman problem).

Continuous, infinite search space characterise functional optimisation problems. Discrete, finite search spaces characterise combinatorial problems. Although a functional problem may have an infinite search space, any computer realisation eventually requires the search space to be discretised to a finite domain. The size of the search space then becomes a function of the desired accuracy, in addition to the number of parameters.

Functional Optimization. There are many approaches to functional optimisation. Differential algorithms use brute force mathematics to derive the minimum of the function. This approach, however, only works when the function is differentiable. For non-differentiable functions, gradient-based or hill-climbing algorithms can be used. A gradient-based approach uses a greedy type algorithm to direct the search in the most promising direction. The greedy algorithm operates on the basis of local decisions to guide the search toward the globally optimal solution. This approach works fine for simple functions, but does not perform as well on complex functions containing many minima. Consequently, a more robust search strategy must be applied to avoid being trapped by local minima. Monte Carlo random search, simulated annealing, and genetic algorithms are search-based algorithms which are applicable to optimization of complex functions.

The search space for a function consists of the domain of the variables contained in the function to be optimised. The solution will be the set of n values, where n is the number of variables in the function, and $f(v_1, v_2, v_3, ..., v_n)$ is either a maximum or minimum solution depending on the type of optimization being performed.

Solution of $f(x_1, x_2, x_3, ..., x_n) = [v_1, v_2, v_3, ..., v_n]$

The application of genetic algorithms to functional optimisation problems involves two things; the encoding of the domain or search space as a genetic sting, and the implementation of the fitness function, which is simply the function to be optimised. T he genetic algorithm evaluates the fitness of each member of a population of strings, and awards an increasing number of copies to stings of above average fitness, while decreasing the overall number of strings with below average fitness. The effects of crossover serve to combine the positive aspects of two strings into one solution. Eventually, the solutions which correspond to the strings in the population, reach a point of optimal or near optimal solution quality.

Combinatorial Optimization. Combinatorial problems attract much attention within the Genetic Algorithm community. The Fifth International Conference on Genetic Algorithms (21) published papers on vehicle routing, traveling salesman problem, and set partitioning. An entire session of the conference focused on scheduling problems. This section presents two combinatorial problems which represent different aspects of combinatorial encodings.

A.4.0.1 Task/Process Assignment. In this problem, m tasks t_i are to be assigned to n processors p_j in a way to minimize completion time of all tasks. Let the string consist of m genes which can take on one of n alleles. Each gene's locus (i) corresponds to a task (t_i) while the gene's value (j) represents the processor (p_j) to which the task is assigned. For illustration, assume a problem of six tasks ($m = 6$) to be assigned to four processors ($n = 4$). A possible encoding is

1 4 3 2 1 4

This encoding states that

t_1 assigned to p_1
t_2 assigned to p_4
t_3 assigned to p_3
t_4 assigned to p_2
t_5 assigned to p_1
t_6 assigned to p_4

The user provides a fitness function to evaluate the encoding and the GA does the rest.

A.4.0.2 Traveling Salesman Problem. Some combinatorial problems present problems to the traditional GA approach (22:170). These problems are characterized as solutions which represent order. A five city traveling salesman problem illustrates the problem with crossover. Let the locus of a gene represent the order in which cities (specified by the value of the gene) are visited. The first gene represent the starting city, the second gene represents the second city visited, ect.... The fifth gene represents the fifth city visited and return to the first city is implied. Let the following strings represent two solutions in a population.

```
Parent 1 = 1 2 3 4 5
Parent 2=  1 3 5 4 2
```

and let the crossover point be 2 (a point between the second and third genes).

```
Parent 1 = 1 2 | 3 4 5
Parent 2=  1 3 | 5 4 2
```

The result of this crossover is

```
Child 1 = 1 2 5 4 2
Child 2=  1 3 3 4 5
```

. Note that both children represent invalid solutions. Child 1 visits city 2 twice and doesn't visit city 3. Child 2 visits city 3 twice and doesn't visit city 2 at all.

Several approaches exist to fix this problem. One approach uses a heuristic to repair any broken children. Another approach uses a penalty function to decrease the fitness of any invalid children.

A.5 Computer Program Development

This section provides pseudo code based on appendix C of Goldberg's book (22). That appendix provides Pascal code for a simple genetic algorithm. Some simplifications where made for clarity: statistical reporting was eliminated and the data structure was modified to more resemble an object-oriented design. Note that the GENESIS code widely used is slightly different in implementation.

Data Definition.

```
  allele-type:        boolean
  chromosome-type:    array[1..max-length] of allele
  fitness-type:       real
  individual-type:
    record of
      chromosome:       chromosome-type
      fitness:          fitness-type
    end record
  pop-type:
    record of
      individual:       array[1..pop-size] of individuals
      sum-fitness:      real
      size:             integer
      chromosome-size:  integer
    end record
```

Control Loop.

```
  pop:          pop-type
  old-pop:      pop-type
  mate1:        chromosome-type
  mate2:        chromosome-type
  child1:       chromosome-type
  child2:       chromosome-type
BEGIN
  USER SETS pop.size, pop.chromosome-size
  initialize(pop)
  FOR EACH GENERATION
    old-pop= pop
    SET pop.size= 0
    WHILE pop.size < old-pop.size
      mate1= select(pop)
      mate2= select(pop)
      crossover(mate1,mate2,child1,child2)
      pop.individual[pop.popsize]=   child1
      pop.individual[pop.popsize+1]= child2
      pop.size=  popsize+2
    ENDWHILE
  NEXT GENERATION
END
```

Evaluation.

```
input/output
  individual:  individual-type
BEGIN
  individual.fitness= FITNESS(individual.chromosome)
END
```

Initialize.

```
input/output
  pop:  pop-type
BEGIN
  INITIALIZE i=0
  FOR i= 1 to pop.size
    FOR j= 1 to  pop.chromosome-size
      SET pop.individual[i].chromosome[j]= RANDOM(0,1)
    NEXT j
  NEXT i
END
```

Select.

```
inp[illegible]
  po[illegible]               pop-type
  t[illegible] itness         fitness-type
  s[illegible]                fitness type
outp[illegible]
  indiv[illegible]al:         individual-type
BEGIN
  SET target-f[illegible]ess= RANDOM(0..pop.sum-fitness)
  INITIA[illegible]ZE [illegible] = 0
  INITIAL[illegible]ZE [illegible]
  WHILE slot < target-fitness
    i= i + 1
    slot= slot + pop.individual[i].fitness
  ENDWHILE
  RETURN(pop.individual[i])
END
```

Crossover.

```
  input
    pop:       pop-type
    mate1:     chromosome-type
    mate2:     chromosome-type
  output
    child1:    chromosome-type
    child2:    chromosome-type
BEGIN
  IF CROSSOVER DESIRED (BASED ON CROSSOVER PROBABLITY)
    SET crosspoint= RANDOM(1..pop.chromosome-length -1)
  ELSE
    SET crosspoint= pop.chromosome-length
  ENDIF
  FOR j=1 to crosspoint
    child1.chromosome[j]= mutation(mate1.chromosome[j])
    child2.chromosome[j]= mutation(mate2.chromosome[j])
  NEXT crosspoint
  FOR j= crosspoint+1 to pop.chromosome-length
    child1.chromosome[j]= mutation(mate2.chromosome[j])
    child2.chromosome[j]= mutation(mate1.chromosome[j])
  NEXT j
  evaluate(child1)
  evaluate(child2)
END
```

Mutation.

```
  input/output
    allele:  allele-type
BEGIN
  IF MUTATION DESIRED (BASED ON MUTATION PROBABLITY)
    allele= not allele
  ELSE
    allele= allele
  ENDIF
  RETURN(allele)
END
```

A.6 Software Available

Software packages described in this section comes from a compilation in *Parallel Genetic Algorithms: Theory and Applications* (18). These packages are available on thor (the parallel network) under

/usr/genetic/Software.

GENESIS. GENESIS (GENEtic Search Implem ıtatioı System) was written by John Grefenstette in 1981. Since 1985 it has been widely distributed in the research community. It is written in C and has been implemented on Sun Sparc Stations and IBM compatible PCs.

OOGA. OOGA (Object-Oriented Genetic Algorithm) was written by Lawrence Davis to support his *Handbook of Genetic Algorithms* (10). It is written in LISP.

Splicer. Splicer was written for NASA/Johnson Space Center. It is written in C and developed on Apple Macintosh. It has been ported to Sun Sparc workstations using X-windows interface.

A.7 Other evolutionary-based methods

Here we briefly mention some other methods based on evolution that the reader should be aw ıre of.

imulated annealing operates very similar to genetic algorithms, but use only one population m:mber and an *annealing schedule* to control the probability distribution used to assign new values to the member. In brief, a paramter called *mperature* is used to determine a likelyhood that the evolving value may take on a worse value, based on its fitness function, than it currently has. As in the settling of molecules in real metal annealing (where the process has its origins), large jumps to higher-energy arrangements of molecules (worse fitness) are allowed when the temperature (time to go) is high. As the system cools (evolves), however, smaller jumps in the less organised direction (to worse fitness) are allowed, until the system finally reaches a cooled (evolved) state. The key to the success of this method is the proper choice of annealing (cooling) schedule. See either Davis or Muruisabal for more on annealing(52:36)

Evolutionary search systems have also been derived that depend more on evolving multiple populations of solutions using mutation as the primary force of change. The reader is referred to the latest ICGA proceedings for more on this new technique.

A.8 Machine Learning and Genetic Algorithms

Discovery learning is a form of learning that focuses on observation and experimentation (12). The key difference between this form of learning and, say, guided discovery or learning from examples is that in discovery learning no specific learning guidelines are given to the learning system. The system must determine for itself what is an example or useful event to observe and what is not. While learning by observation is simply trying to interpret what is happening in the environment, learning by experimentation allows the system to interact with the environment, try things, and see the results. Discovery learning includes both of these methods.

The advantage of this type of learning is that its autonomous, i.e. can be done with no outside guidance. This means that a system using this learning approach can be trained by just "plopping" it into the environment and letting it explore. An autonomous opponent simulation with this ability could potentially learn to dog fight and drop bombs by observing these activities and experimenting with the possible actions allowed to it (banking left, activating the bomb release, etc.).

Genetics based machine learning uses *genetic algorithms* to guide the construction and restructuring of knowledge representations and allow a system to adapt to a (possibly changing) environment.

The recent advances in genetics-based learning has opened the possibility of using these techniques as part of real-time systems that require some form of machine learning capability. Experimental results in many applications areas show that genetics-based learning systems can relatively quickly adopt to simpler (typically single task) learning situations (12, 22). However,

little work has been done applying these techniques to more complex applications (40:9–10), though some experimentation has been done (for example, (27, 36, 77)).

One method of managing complexity is by using *multi-agent systems* (MASs). These are systems that contain multiple *agents*, autonomous systems that react to their environment. Agents can be intelligent (incorporating AI techniques) or non-intelligent, and can cooperate to varying degrees with the other agents in the system. Multiple agent systems allow complex adaptive tasks to be broken up into smaller tasks that use specialised agents to perform functions such as route planning and sensor processing, thus managing the overall complexity of the system. (24)

A.8.1 Current Genetics-base Machine Learning Techniques. There are four basic approaches to using genetic algorithms in machine learning applications: parameter-based methods, "Michigan" classifiers, "Pitt" classifers, and evolutionary programming (12).

Parameter-based techniques

Parameter-based techniques operate by defining a set of function parameters that describe the particular problem to be learned. Typically these parameters describe various controls that can be "tuned" (adjusted) for optimal performance. This technique is very similar to the typical genetic algorithm application, where the algorithm searches for a best (or near best) solution to the problem. Here, the fitness function is external to the learning system, however, and the algorithm must find a set of parameters that allows the controlled function to give an adequate response (as determined by the environment) for any environmental input. Since the varying of one parameter can affect the correctness of other parameters, finding a setting of all parameters that perform well for all allowed inputs is a combinatoric problem tending towards NP-complete[2]. See pages 61 – 101

[2] A combinatoric problem is one that requires an increasing number of calculations to solve as the size of the input data grows. NP-complete problems are those combinatoric problems that generally take a deterministic search algorithm an exponentially increasing amount of time to solve as the amount of input data increases. These are the hardest search problems to solve due to the extensive amount of computer time needed to examine the exponentially growing number of possible solutions and are virtually unsolvable for large numbers of input parameters, unless some form of heuristic (a method for deciding which of many alternatives is most promising (54:3)) is used (57:40) (55:343–344) (9:916–963).

and 104 – 108 in *The Handook of Genetic Algorithms* by Davis (10) and the various applications, especially the robot trajectory example (10:144–165), in Part II of this reference for more on this method, including examples of its use.

Classifiers.

Classifiers are systems which receive input *messages* (encoded data strings) from the environment, match these messages against the *condition* part of rules (*classifiers*) in a rule set, and generate some *action* if a rule's condition string is matched. There are currently two general approaches to classifier system design, the "Michigan" approach and the "Pitt" approach (12). These are each described below.

"Michigan" classifiers.

The "Michigan" approach was developed at the University of Michigan as part of John Holland's efforts to adapt genetic algorithms to machine learning applications (12). In this approach, the genetic population represents a set of rules, each composed of a condition part and an action part. The condition part consists of one or more (depending on the design) patterns (conditions) that are compared against the input *messages* from the environment. If only one rule's condition part is satisfied, then that rule's action part is generally allowed to fire (execute). If multiple rules match, then typically some form of arbitration takes place, such as a bidding auction where rules bid a fraction of their current fitness. One (or more, depending on the design) of the winning rules then get to fire their action parts. Actions can either generate outputs to the environment or generate new messages, which are placed on the *message list* with any new messages from the environment and can trigger other rules as if these new messages came from the environment. This sequential triggering of such *coupled* rules is called *chaining* and allows an environmental input to successively trigger multiple rules and output complex responses over time (12) (36:chapter 4).

"Michigan" classifiers learn by establishing rules to handle the various environmental inputs. A mechanism called the *bucket brigade*, where rules that fire pay a portion of their fitness to

the originator of the message that triggered them, allows *payoff* from the environment for a good response to be passed back (indirectly) to those that started the successful chain of rule firings. Rules that don't result in payoff or in another rule firing will slowly lose their strength (from bidding to fire, etc.) and will die out. In this way, a chain of rules that doesn't result in eventual payment will be starved out rule by rule. This process lets the successful coupled chains grow stronger (and more likely to be selected to provide the system's output) and the less successful rules and chains to die out (12, 36).

Classifier condition parts typically are binary strings of ones and zeros, which are mostly encodings of the solution parameters into one string. To facilitate some generalization in the matchings, a wildcard symbol is used to represent either a one or a zero. This allows a single classifier to match multiple input messages and allows a generalization of the rules to take place as the system learns. Wildcarding also allows the building of *default heirarchies* (12)(22:247–254). In a default heirarchy, different rules have different levels of generality (*specificity*). More general rules will tend to match more input messages. The action parts of these rules, however, is generally not the appropriate response to all of these input messages, so more specific rules emerge to cover the exceptions. This leveling of general and specific rules is cited by some as key to the learning of complex information (36:34–36, also 190–220).

New rules are induced into the 'Michigan" classifier using various chain-forming techniques as well as a genetic algorithm. The genetic algorithm, in particular, looks at the pieces of successful (high fitness) rules and builds new children rules from these building blocks. These rules replace the less fit of the population, thus increasing the potential overall fitness of the rule population and the overall performance of the classifier system. Thus, the genetic algorithm provides the major learning mechanism of "Michigan" classifier systems (12).

"Pitt" classifiers

"Pittsburgh" or "Pitt" classifiers operate a bit differently than the above "Michigan" systems. In "Pitt" classifiers, each population member represents not a single rule, but an entire rule set encoded into the string. These classifier systems work by setting up a population of possible rule sets. This population is then tested, member by member, by running each rule set through an evaluation function that provides a set of environmental inputs and tests the outputs for correctness. Each rule set is given a fitness rating from these tests. Then the genetic algorithm is used to breed the population of rule sets, with the children of better performing rule sets replacing the less fit rule sets. In this way, rule sets of high fitness (i.e. best adapted to the environment) emerge.

"Pitt" classifiers optimize rule sets in a way similar to that used by standard genetic algorithms to optimize function values. This approach has many consequences. First, the adaptation only takes place when the genetic algorithm is run, and not while the rule set is actually being used. This makes it difficult to judge the usefulness of individual rules in a rule set. Second, this approach does not have any direct method of forming default heirarchies or coupled rules, so the rules thus generated tend to be specific to the test cases provided by the tester during rule set evaluation. Both of these tend to result in "Pitt" classifiers evolving *brittle* rule sets (12, 34), i.e. rule sets that are optimized for the test cases used to train the system, but that tend to perform less successfully on related but unlearned cases and also on noisy inputs. Research continues in this area (3).

Evolutionary programming

Evolutionary programming techniques are similar to the rule-based classifier systems, but instead manipulate parts of programs instead of parts of rules. Most research in this area has taken the evolve and try approach (12), which is similar to the "Pitt" classifier approach. Current work includes Koza's work with LISP expression manipulation (41, 42). This approach tends to suffer from the same brittleness characteristics that affect the "Pitt" classifier systems.

A.8.2 Current Trends in Genetics-Based Learning. Current trends in the field of Genetics-based Learning include adapting the basic classifier systems to better address the problems asso-

ciated with them, and forming hybrid systems with multiple genetics-based components or with genetics-based components as just one of many agents in the system.

Most work at improving classifiers has focused on the "Michigan" systems. Efforts include adding triggered rule discovery and rule clustering via Booker's GOFER system (72:265–274), modifying the bucket brigade (72:311–216), adding variables (72:334–339) and fuzzy constructs (3:346-353) to the rule systems, and others (72, 3). Riolo's work at adding hypothetical states to "Michigan" classifiers (47:316–326) and Muruzabal's work with a database scanning Adaptive Predictive System application (52) are some of many examples addressing the "forgetfulness" of these systems.

Hybrid systems have also emerged, including Grefenstette's SAMUAL system (27) and Dorigo's multi-layer classifier systems (15). Work with *animats*, software representations of simple living creatures, have generated many new approaches to adapting learning systems, including genetic algorithms, to the task of adapting to the outside environment (47). Multi-agent systems have gained new interest, including the work by (24) and others. Merging genetics-based approaches with neural network systems is also attracting much research effort and shows promise (10:202–221) (51).

In addition, Holland, et al., suggests many other ways that classifier systems can be modifier that have yet to be fully researched (36). Much remains to be done in the field of genetics-based Learning.

Appendix B. Classifier System Basics

This appendix introduces the reader to the basics of classifier systems. This material was already introduced to some extent in Chapter 3, but it was felt that a more comprehensive introduction was needed to provide the needed background for Chapter 4. The reader already familier with classifiers should find no surprises here.

Key issues in classifier construction include the architecture used, the encoding of the search and solution spaces onto the genotype, and how selective pressure can be used to promote adaptive learning. The reader is referred to Goldberg's book for a good introduction to these topics(22).

A short review of the developments in genetics-based classifier systems is provided in Appendix C.

B.1 Basic Classifiers

Classifier systems or, more simply, *classifiers* are rule-based systems that maintain a population of prioritized rules that fire when their condition field matches that of an incoming message from the outside environment(35:173). The basic parts of a classifier are its input interface, a Message List, a Rule List, and an Output Interface. Each of these components, and their interaction, are described below.

The *input interface* provides the "eyes and ears" of the classifier system in that it fully defines the perceived world of the classifier system[1]. A set of *detectors* is used to convert input messages from some outside representation to an internal encoding. This encoding typically takes the form of a binary bit string (using the alphabet { 0, 1 }) called a *message*. The messages so produced by the active detectors in the system enter their messages onto the message list where they await processing during the next classifying cycle. See Figure B.1 below.

[1]This is not fully correct, since, as will be seen later, the classifier system itself can supply inputs that look just like those from the outside world and so act on them. However, it is correct enough for the current discussion.

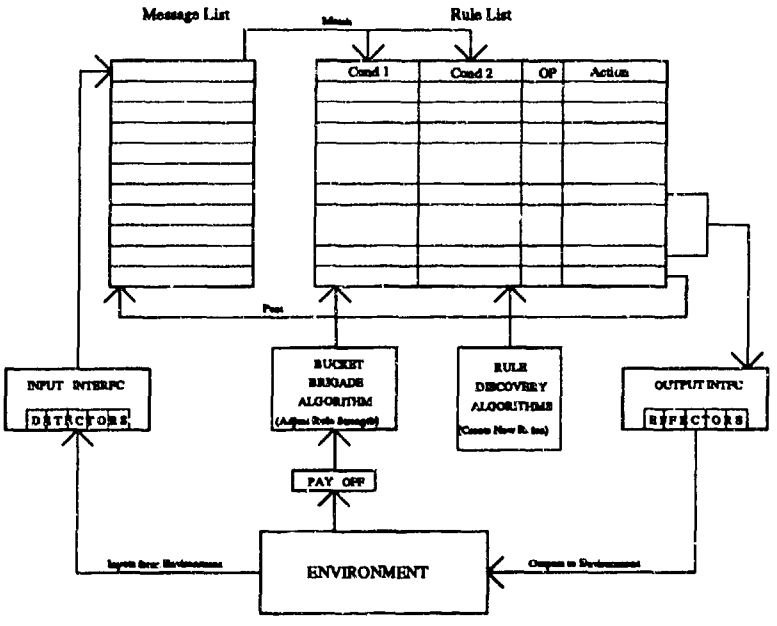


Figure B.1 A Standard Classifier System.

The *message list*, in the simple case, now contains messages that are the same length (in bits) as the condition strings in the Rule List. These conditions, which use a more complex alphabet with three symbols: { 0, 1, # }. The first two are as in the messages and represent the binary encoding. The third symbol represents the "don't care" condition, meaning that a match is made if the message has *either* a 0 or 1 in that bit position. Thus a single condition string (field) can potentially match many messages at the same time.

The rules (also known as *classifiers* since they classify the input messages into categories or concepts – see Chapter 2) are composed of a set of one or more condition fields that form the "if" part of the rule, and an action field that performs some action when the rule is fired. The condition fields are generally the same size as the input messages, and are of the same form, except for the don't cares. The action field can be any size, but generally is also the same size as the messages, for reasons detailed below. Thus the if-then rule

> IF there is (a message from a detector) indicating food to the left,
> THEN (by issuing an effector message) tell the output interface to turn left.

can be encoded as the classifier (rule)

```
#### ##10 ####/#### #### ##11
```

where the 10 in the condition field (first string up to the slash (/)) encodes the fact that there is food to the left (in this example and encoding), and the 11 in the action field (the symbols to the right of the slash) is the encoding used here to tell the output interface to turn left. Note that the encoding used and the positions of the bits in the strings, as well as the string sizes, are all changeable and, in fact, can change while the system is executing.

Rules in the Rule List where all conditions match at least one message are candidates for posting messages to the output interface. Many rules can be active at once (i.e. can fire simultaneously during a classifier cycle), but a limit is generally imposed to limit the use of computational resources, among other things[2]. Note that all condition fields must be matched by some message for a classifier to become active. Thus if more than one condition field is used, these fields act as the logical ANDing of matched classifications. Rule condition fields with don't care bits, on the other hand, act as logical OR constructs, allowing multiple possible messages to match the condition. When one throws in a negation operator, represented as a minus sign preceeding the second condition field in the example below, then the resulting classifier system can be shown to be computationally complete[3](35:175).

```
001100##0, -1######## / 0000111##
```

One further note is necessary. The don't care bits in the action act as "pass through" bits; i.e. the bits in those positions of the message that matched the first condition field[4] appear in those positions of the resulting action message. This allows the passing of information from the input of the rule to the output. Thus the above message states

[2]These other things are discussed shortly.
[3]A negated classifier matches if NO message in the message list matches it.
[4]By convention the first condition field is used, but any field might be used to provide the pass through bits.

IF a message exists on the Message List that starts 001100 and ends with 0,
AND NO message exists on the Message List that starts with 1,
THEN generate the message string starting with 0000111 and ending with the last two bits of the message that matched condition field one.

In simpler applications that don't require rule chaining, such as stimulus-response systems, the action field need not be of any particular size, or even in a binary form. Learning to predict the output of an unknown boolean function given its inputs is one example of such an application(22).

Rules that fire can either send a message to the output interface to activate an *effector* to change some aspect of the environment, or they can generate an "internal" message that is placed on the input Message List to be matched against the next cycle. There is no difference between these two types of messages (in the general case), except in the encoding (specific bit patterns are matched by each effector interface just like each rule has at least one condition field that must be matched against a message in the message List before the rule can fire). This feedback of messages back on to the message list allows the system to "remember" a message from one cycle to the following cycle. This is important since what is really being passed onward to the next cycle is the fact that a particular concept was recognized by the system on the previous (now current) cycle. This use of messages is looked at more closely when rule chains are discussed below.

B.2 Strength, Specificity, and Default Heirarchies

When many rules match, which rules to fire is determined by a competition (*auction*) based on the *bids* of the rules. This bid can be based on many factors, including the rule's strength (as determined via some credit allocation mechanism (discussed next)) and its *specificity*.

A rule's specificity is a measure of how specific a rule is and is generally related to the number of "don't care" bits in the rule. Specificity for a rule with multiple condition fields is usually taken as the average specificity of the conditions. Thus

$$specificity = (S(cond1) + S(cond2) \ / \ (total\_bits\_in\_all\_conditions)$$

where $S(cond)$ is the total number of 1's and 0's in the condition (i.e. bits that are not #). For not-match conditions, S is the number of #'s in the field, since these restrict matching to more specific bit strings.

Rules that have a higher specificity tend to match more specific conditions in the environment (via the detector messages), while those with lower specificity match more environmental states. This can lead to ***default hierarchies*** where a set of rules handle differing levels of specificity. For example, for the rules

```
Rule 1:    #### ##01 11## / action 1
   and
Rule2:     0011 0101 1110 / action 2
```

the first rule matches more cases (states) than the second. If Rule 2 produces higher bids in this situation, then action 2 will get selected when Rule 2 applies, but will allow Rule 1's action 1 to execute when it doesn't. If we translate the above rules to mean:

```
Rule 1:    If (the light is yellow) then (slow to a stop)
   and
Rule 2:    If (the light is yellow) and (in hurry) then (speed up)
```

then, if the more specific condition is met, we execute action 2, else we use action 1. This handling of exceptions is the key to default hierarchies and allows concepts to be represented in a minimum of rules. Specifity is generally used in generating the bids of classifiers, as discussed below.

B.3 Credit allocation in classifiers

In a sense, the credit allocation algorithms are learning algorithms since they adaptively adjust the firing salience of rules so that the rules that best categorize a particular concept seen by the system get first crack at later presentations of the concept. However, since no new rules directly result from the credit allocation mechanisms, we define the discovery operators to be those operators that directly add and remove rules from the population.

The idea behind *credit allocation* or the distributing of payoff received by one classifier to others, is to increase the strengths of rules that act as *stage setters*. Stage setters are rules whose actions set up conditions for other rules to fire. Proper credit allocation can lead to chains of rules that fire in adapted sequences in synch with the demands of the environment.

The em bucket brigade algorithm is one method of handling the credit assignment task and is used with the majority of classifier systems today(72, 3). The bucket brigade uses an analogy to a service economy to pass payment back to immediately preceeding rules that allowed the current rule to fire(36:72). If a rule makes a profit (receives a payoff from the environmnet), it passes part of this payment to the rules (or detectors) responsible for producing the messages that allowed it to be activated. Actually, the algorithm is implemented using a system of taxes and payments such that a minimal amount of information is needed to properly allocate strength.

Bucket Brigade. The bucket brigade operates as follows(59)

- Each cycle, each classifier is matched against the messages on the message list. Those that match post a *bid* (a fraction of their total fitness) as a payment to participate in the auction to follow. This bid is subtracted from the fitness of the classifier and divided among the detectors and rules that produced the messages that allowed the classifer to fire. Those receiving payment add it to their total fitness.
- If the rule then receives payment from either the environment (for a "useful" action) or from other rules that fire later, this pyament is added to this rule's total fitness. If no payment is received, then the rule is assumed to have not generated any useful message this cycle and therefore its total fitness ends up reduced by the amount of the bid.

Note that rules that never generate any useful action eventually "starve" in this approach, where as those that at least occasionally generate useful messages reach a *fixed point* (in a static situation) that reflects the average of the payments received times a factor dependent on the fraction bid. This value approaches:

$$S_{fp} = (R + P)/(k * BidRatio)$$

where R is the average *rewards* received (per activation) by the rule, P is the average of *payments* received, k is a constant and *BidRatio* is the fraction of the total strength bid each time the classifier is activated.

Note that the actual fixed point depends on the average number of rules that use the classifier's generated messages[5] and on the number of rules that must be paid off[6].

The bidding process can be further modified by adding various *taxes* to the bidding process. For instance, in the operation of the system, rules may be generated (via the discovery algorithms discussed later) that never bid but never match any messages. These classifiers would remain in the rule population taking up space and (assuming a limited rule list size) would take up valuable rule space. A *head tax* can be used to take a very small fraction of the total fitness away from *all* classifiers each cycle, thus eventually dropping the fitness of these rules to a point where the discovery algorithms will look at them for replacement by new rules they generate.

Another problem that arises is when many rules fire at once, but only a limited number of actions can be accomplished in one cycle. In this case a competition (*auction*) can be held to choose the rule that will generate the action for that round. In this case the non-producing rules those that matched but lost the competition) can either have their bid taken subtracted (and distributed) or just be removed from the match list as if they hadn't bid. Both approaches are possible can be used.

Yet another problem is what to do about detector messages. These messages do not come from other rules, since they are generated by the environmental interface, and so paying this source effectively removes credit (fitness) from the rule population, and away from rules that generated messages. Some systems (such as CFSC-1) allow the detectors to receive a smaller fraction of

[5] In the general case classifiers can generate many messages on each activation. In practice, however, only one message per activation is allowed to prevent one classifier from "taking over" the rule population.

[6] If a full bid fraction is given to each message source. Another alternative is to divide the bid among the different sources.

the bid (down to 0 %) to promote rules that generate useful messages. However, the use of this type of controlled distribution must be looked at carefully, since this transfer of payment promotes internal generation of stimulus at the expense of triggering off of environmental stimulus. An alternative (and the one chosen by the PPLS implementation discussed later) is to prevent rules from generating such ***hallucination*** rules that mimmic detector messages. This is done by ***tagging*** messages as to their source and is looked at more below.

The taxes can be further modified to favor stronger (more fit) or weaker rules when both are competing for the right to post messages. One way is to take the bid of each rule and raise it to some power *BidPower*. If *BidPower* is greater than one, then above average fitness rules[7] generate greater effective bids and below average rules generate bids lower than they normally would. The opposite happens when *BidPower* is less than one. In the limiting case where *BidPower* is actually zero, then selection will be random, with strength having no effect on the effective bid. *BidPower* is set to 1.0 in most systems. Note that each tax can have a *BidPower*, allowing great versatility in how rules are credited.

The bucket brigade is not without troubles, however. Many modifications to the taxing structure (some noted previously) are generally needed to build *chains* of rules that can handle time sequences of behavior[8]. Since payment is awarded on an incremental basis and is only a fraction of the average a source's fitness, rules farther back in a chain tend to receive a smaller and smaller part of the reward. This is because the reward is generally divided between many producers at each time step, and each rule only gets part of it. Thus the reward gets spread thinner and thinner as each rule in the chain pays to not only the previous rule in the chain, but any other rules active at that point in the chain. *Bridging classifiers*, or rules that trigger on the reward and

[7] For simplicity, the bids are generally normalized by the average fitness of the rule population. Thus the bid of a classifier is actually $bid = (strength\ /\ population.strength)^{BidPower} * BidRatio$. This bid can be modified by multiplying the *strength* by the *specificity* of the rule to promote default hierarchy formation. Thus $bid = (strength * specificity\ /\ population.strength)^{BidPower} * BidRatio$. Note that only $strength * BidRatio$ is subtracted from the rule as the actual tax.

[8] Chains are discussed momentarilly.

pass payment back to the beginning of the chain, can develop to alleviate part of this problem(58). However, such rules don't easily form in practice[9] and better mechanisms may be needed to allow chains to form easily and naturally (i.e. on their own).

In summary, the bucket brigade provides an incremental (*time difference*) method of credit allocation that requires no long-term tracking of rule activations to be effective. Through a sequence of taxes and payments, rules gravitate to steady-state values of fitness proportional to their role in allowing the agent to reach a payoff state. To date this is the most used method to handle the credit allocation problem in standard classifiers[10] and is the method used in the PPLS system addressed later in Chapter 4.

Other credit allocation methods. Other methods of credit allocation were addressed in Chapter 2 and are used in many systems. For example, a profit sharing mechanism can be used where the fitnesses of rules are only modified at the end of an *epoch* (usually defined as the cycle where a reward is received). This type off approach was used by Holland and Reitman in the original CS-1 system and later Grefenstette used a modified version he termed a Profit Sharing Plan (PSP)(26). This method requires the tracking of all rules that have been active during the epoch. When payment is received, it is distributed to all rules that were active and so assumed to have some say in the outcome leading to the received payment. Generally, many trials are needed in such systems to distribute payment fairly to all rules, since both "good" (effective) and "bad" (disruptive) rules receive a share of the payment, regardless of their function in getting it. Many trials allows rules that don't contribute the chance not to fire and eventually lose fitness and fail to bid at all.

[9] This might be because of a lack of evolutionary pressure forcing such bridges to form. The likelyhood that a rule would develop that triggers off a specific rule earlier in a chain and is active (bidding) during the payment of reward is rather low, unless a situation exists that keeps the rule active throughout the epoch. But in this case the epoch must have some defining signature (partial state representation) that the rule can detect and match against (such as a bit in a detector message structure indicating that an enemy is being evaded). Note also that the existence of such bridging rules remove strength from the other rules actually doing the work (producing effective messages) during the epoch.

[10] Pitt systems, since they use an off-line epoch-based approach at interacting with the environment, tend to use other allocation methods. See below

Another problem with this approach is the need to determine when an epoch starts and ends. The end is usually assumed to be when payment is received, but this doesn't account for incremental payments that may be received along the way to reaching some goal state. The start is more difficult to pin down, since it is not always obvious when the actions that lead to a goal actually started. For these reasons, this method is usually only used in systems that can discern a distinct start and end to the learning period, such as those learning a specific maneuver for a controlled aircraft, given a specific set of (possibly varying) starting conditions but a specific goal (fitness measure).

B.4 Support and Actions

Actions in a classifier system can either be direct consequences of a rule firing, such as is done in many other rule-based systems, or can be indirectly triggered via the generation of messages that direct them. This latter approach is used in the standard classifier system of Holland(36) and is described here.

In this approach, the environment is manipulated by the classifier system via a set of *effectors* which can be considered subagents triggered by system messages that perform specific tasks in the environment. Rules trigger effectors by generating *effector messages* that match the conditions required for the effector to fire. Feedback from the effector (if any) comes back to the system via the *detectors*.

When multiple messages reach an effector simultaneously, the effector must determine which of the proposed actions it is to do. This process of *effector resolution* is handled by building up *support* for each action based on the *intensities* (strengths) of the messages that are received. Each received message has an intensity based on either the bid of the rule that posted it or, in the case of detector-generated messages, the value supplied by the detector. (Detector intensities generally indicate the "urgency" of the message, as determined by the environment.) The system adds th

support for each action. The action that has the most support is chosen as the action to perform on that cycle.

B.5 Chains and tags in classifiers

Tags are sections of condition and action fields reserved for representing context information. A tag is a bit pattern that prevents a condition from firing unless the bits in the message match those of the tag field. Tags essentially divide the rule population into groups that can address subtasks of a problem.

Chains form when one rule's action is tagged to match the tag in the condition field of another rule. The tag could be a partial match, requiring other conditions to be met before the second rule could fire, or it might be a full match of the field leading to a reflex type of chaining. These are both shown below (spaces added for clarity):

```
Message:    1010 1001

Rule 1:     #### 1001 / #### 0111    generates message:  1010 0111
Rule 2:     1010 0111 / 1000 1111    triggers effector:  1000 with action 1111

Rule 3:     1010 1##1 / 0000 1111
Rule 4:     0000 1111 / 1000 1110    triggers effector:  1000 with action 1110
```

In the first rule subset, 0111 can be considered the tag. In the second, the entire string 1000 1111 acts as the tag.

If Rule 1 fires, it will generate a message dependent on the messages it matches. In this case, the message it generates matches Rule 2's conditions and so Rule 2 generates a message that activates an effector that responds to the 1000 prefix.

In the second example, Rule 3 fires and generates the fully specified string 00001111. This matches the reflex rule, Rule 4, and it triggers effector 1000 with action code 1110. This is a reflex response since firing Rule 3 on one cycle guarantees (barring rule competition) that Rule 4 will fire the following cycle.

Chains provide classifiers a way to represent seqences of actions that lead to a goal (payoff). Although they have been shown to form in some situations(63), discovering them may take many cycles (and many trials and errors). Because of this, other methods have been looked at to generate sequences of actions to address environmental needs, which we look at later in this chapter.

B.6 Review of non-discovery classifier operation

To summarize, a classifier system steps through the following operations each cycle:

1. Read in the messages from environmental detectors and place them on the message list. Note that other messages generated by classifiers the previous cycle may already be on the message list.
2. Perform a match of all rule conditions to the messages on the message list. Mark those that match as eligible for producing action messages. Generate bids for all marked rules. Apply any bid taxes.
3. If too many messages would be generated, conduct an auction based on rule bids and select the rules to fire. Mark the winners as producer rules and generate action messages. Apply any producer taxes.
4. Perform effector resolution so that all actions generated are consistent with each other. Rules that have their actions voided are remarked as non-producers.
5. Execute the actions by sending the messages to the effector interface. Triggered effectors perform the specified action in the environment.
6. Clear the message list and post non-effector messages to the new list to match against on the next cycle.
7. Apply any head tax to all population members.
8. Repeat the cycle.

This loop continues until the system is stopped by the user or the system "dies" (is no longer considered active in the environment).

Appendix C. A short history of classifier systems

This Appendix presents a short history of classifier systems. We do this to provide the reader with the basic issues of classifier systems in a more or less time sequencial form. Though a historical perspective, many new concepts are introduced. The reader should be able to skim this section, however, since those ideas used later are mostly redescribed when again used. Most of the material here is from the article of Wilson and Goldberg(82).

C.1 Historical Review

Classifiers have their beginnings in the early work of John Holland. In his paper "Processing and Processors for Schemata," published in 1971, he started to progressively modify the classifier concept and its structure until, in his 1975 hallmark book "Adaptation in Natural and Artificial Systems," he presented a rule-based system known as the *broadcast language*(33:141). This precursor to classifier systems had most of the qualities of classifiers as they are known today, but with the differnce that a "broadcast unit" (the equivalent of a rule or classifier) could directly create other broadcast units, while classifiers cannot(35:172). Classifiers evolved in Holland's 1976 works and reached a somewhat standard form in his 1980 "Adaptive Algorithms for Discovering and Using General Patterns in Growing Knowledge-Bases."

The first classifier system was the Cognitive System One (CS-1) developed by Holland and Reitman and published in 1978. This system was able to run a simulated one-dimensional maze with payoff at the two ends. Each end had a different resource (food at one end, water at the other) and the system had to learn which way to step depending on its current needs(75:139). Instead of the *bucket brigade* (addressed in the next chapter), the system divvied out reward to those classifiers that had been active during an *epoch*, which ended when a resource was reached and consumed. CS-1 successfully demonstrated learned behavior within this environment.

Steven Smith took a different approach in his dissertation of 1980 entitled "A Learning System Based on Genetic Adaptive Algorithms"(75). In what was to become the representative "Pittsburgh" approach, his system stripped away the credit assignment mechanism of CS-1 and, instead, used genetic operations on strings encoding entire rule sets to build his LS-1 (Learning System One) system. By competing rule sets against each other, he was able to genetically search out complex patterns of behavior via evolutionary pressure. He tested his system against Waterman's poker playing task (one specific version that had some learning capability) and showed remarkable success by showing that his system could learn by doing and, after a time, consistently beat Waterman's more complex[1] system. As with most classifier systems, a feedback function providing a measure of success or failure was the only link Smith used to judge the fitness of the system's actions.

This split from normal thought lead to the two camps of classifier theory, they being the "Michigan" camp and the "Pitt" camp after the locations (University of Michigan and University of Pittsburgh) that they were first developed at. More on this later.

Lashon Booker, in his 1982 dissertation based on the standard classifier approach, addressed many of the issues that were becoming apparent in classifier system research(4). He argued that the previous policy of matching condition fields "all or nothing" lead to much genetic material being wasted in those rules that are almost perfect matches. He shows that changing the match score to account for close matches (based on the number of alleles (bit positions) that matched) allows his system to better adapt to his environment. He also introduced a *restricted-mating policy* where genetic operations selected from those classifiers that were active recently instead of the entire rule population, and the payoff, when received from the environment, was distributed to the active classifiers. The idea is that, assuming the system is trying to address a specific environmental state, that the active classifiers represented a closer approximation to the classifiers needed than

[1] In requiring much more feedback from the environment

any others in the rule list. This type of restriction leads to clusters of classifiers that fill *niches* in the environment. Whenever a state was encountered that no classifier sufficiently matched, the system would draw from the entire population and try to form a new niche(4).

Goldberg in 1983 applied a classifier system to the dynamic control of a gas pipeline(22). His system showed the emergence of *default heirarchies*, levels of general and specific rules that allow a task to be specified more concisely. He also introduced the *noisy auction*(82) as an alternative to *roulette wheel selection* where rules are selected for mating based on the probability formed by dividing their *strength* (a measure of the rule's utility) by the total strength of the rule population.

Wilson in 1985 presented another use of the standard classifier system, that of controlling an artificial animal (an *animat*) in a simple simulated environment. This is one of the first uses of cover operators to create rules that fill gaps when no appropriate rule exists in the system. It is also the first example of a bucket brigade-like algorithm operating under intermittent payoff (i.e. when payoff only comes once in a while)(80).

Forrest in 1985 looked at the use of classifer systems in implementing a subset of the KL-ONE semantic net language(20). She showed that symbolic representations can be supported and implemented in classifier systems. However, it is still unclear whether such structures can evolve usefully under genetic pressure.

Rick Riolo has addressed many of the problems with classifier systems, including long chain development (where a number of rules linked by action messages fire one after another to generate a sequence of behaviors) and default heirarchy development(63). He showed that *bridging classifiers* can be used to carry strength down a classifier chain, as Holland predicted in 1985. He also developed a "standard" classifier system known as CFS-C (Classifier System in "C"), which is now available as public domain(59).

Wilson in 1987 published work on a system known as BOOLE that was a single-step classifier (no rule chaining) and was able to classify boolean logic functions. He also demonstrated parametric

control of some system parameters. Sen in 1988 later modified this system to achieve learning rates better than connectionist networks on the same problem(82).

Robertson and Riolo in 1988 investigated the letter sequence problem in detail. This problem involves the guessing of what letter will come next, given a previously seen set of letters. By using *triggered coupling* of rules and other *cover operators*, they were able to build chains that showed the use of internal messages (messages from one classifer's action that are matched by a condition of another rule the next cycle) to successfully predict letters. Though similar to the *create* mechanism that Wilson used in the 1985 Animat problem, much ground-breaking analysis and many obserations were provided. They also did some work with population sizing on this problem. Problems with using the GA in creating (discovering) new rules was also noted, and their work confirmed many of the problems noted by Booker earlier, as well as others involving the various taxes posed on rules to limit their unlimited propagation(58).

Wilson in 1988, looked at several aspects of bidding and payoff, and found that a problem with rules overgeneralising could be avoided by removing the *specificity* bias (a bias toward more general rules, i.e. those with more wild cards) from the bids of classifiers. This fixed many problems, but forced the restricting of the bid payments to only those bidding rules that supported (agreed with) the winning rule(82).

Grefenstette in 1988 presented the first hybrid system that used components from both the "Michigan" and "Pitt" approaches. In his system Rudi, and later in his Samuel system, he used basically a Pitt approach rule set processing system based on a modified version of his Genesis genetic algorithm system. In addition, he used a credit allocation mechanism to track the utility of the rules in each rule set. Instead of the bucket brigade, however, he used a version of the epoch-based system used by Holland and Reiman in their CS-1 system, which he named a *profit sharing plan* (PSP). He showed that this system better distributes an intermittant reward than the bucket brigade algorithm in his limited two-dimensional state space example problem. The credit

assigments are then used to cluster rules that tend to work together closer together in the rule strings, thus increasing the likelyhood that the cluster of related rules would be transferred intact by the crossover operator to the receiving offspring. The results were better in this environment than those using a standard "Michigan" or "Pitt" system alone(26).

C.2 On-line versus Off-line performance

The on-going debate of on-line (incremental) verses off-line (batch) systems is routed in the two approaches used to implement classifier systems: the Michigan and the Pitt approach[2](12).

In brief, the CS-1 type of systems originally proposed by Holland avoid ***brittleness*** by simultaneously developing many alternative classifications for any particular concept and then choosing between them probabilistically based on past performance. If at any time the most fit rule begins to fail, it quickly loses fitness and the next alternative takes its place to be tried. Also, the genetic algorithm can access and manipulate rules individually, based on each rule's past performance, allowing selective generation of potentially better rules.

The main weakness of this approach is the lack of high-level guidance to place selective pressure on the overall performance of the system. Without guidance these systems learn parts of a problem but have problems with the big picture if it's too much more complex than a simple environment. Some methods have been derived to address this (see below) but the perfect fix has yet to be found.

Pittsburgh systems, on the other hand, evolve entire sets of rules, with the selective pressure aimed at overall performance. Though the rule sets may take much longer to evolve (since there is typically no indication of which rules in the rule sets to concentrate on), once a rule set is created that performs well, it generally performs well in all situations it was trained on. And if such training involves a complex system, then the behavior this system requires will be learned.

[2]See Appendix A for more on the issues involved.

The negative aspects of Pitt systems include their slowness, since a population of entire rule sets must be evolved and tested each generation. And since the system learns the desired behavior by processing a set of training examples in a batch fashion, the tendency to overtrain is strong which, in turn, leads to brittleness (the not so graceful degradation of performance) when the system must perform outside the area it was trained in. The concepts of the training set are learned, but not the concepts of the target system. And since each rule in the resultant rule set may do a specific task, when this rule fails the system has nothing to fall back on.

As is discussed in Section 3.4, there are solutions to many of these disparencies; but each generally has a cost. The choice of system to use still largely depends on the end results desired by the user and what the user can tolerate.

Appendix D. Volume II

Volume II contains the following items and is not included in this published volume. These materials can be obtained by email to:

lamont@afit.af.mil

D.1 Phased Pilot Learning System User Manual

This appendix includes a user manual for the Phased Pilot Learning System as well as for the new features of the modified CFSC classifier system.

D.2 CFSC-1 User Manual by Rick Riolo

This appendix contains a copy of the CFSC-1 User Manual that comes with the public domain version of this system.

D.3 Test Cases and Test Results

This appendix presents the detailed parameters and data of the tests in Chapter 6 and presents more detailed results than presented there. Other tests, such as with different discovery learning operators and with the PDP-C system interface, are also presented here. Analysis of these tests is included.

D.4 PPLS Source Code

This appendix contains the entire source code for the system, including the CFSC-1 package of subroutines (as modified) and the interface code and the PDP-C Clips code. The code included is sufficient to implement the test version of the system. All code may be available from the above address. All code is currently implemented for a Sun workstation environment running Unix,

though the test version should be portable to other environments having access to an ANSI C compiler. Access limitations may apply.

Bibliography

1. Agre, Philip E. and David Chapman. "What are Plans For?." ***Designing Autonomous Agents*** edited by Pattie Maes, MIT Press, 1993.
2. Banks, Capt Sheila B. and Major Carl S. Lissa. ***Pilot's Associate: A Cutting Edge Knowledge-Based System Application.*** Technical Report, Department of Defense, 1991 (roughly). Unpublished.
3. Belew, Richard K. and Lashon B. Booker, editors. ***Proceedings of the Fourth International Conference on Genetic Algorithms***, Morgan Kaufmann, 1991.
4. Booker, Lashon B. ***Intelligent Behavior as an Adaptation to the Task Environment.*** PhD dissertation, University of Michigan, 1982.
5. Bruderer, Erhard and Alex Shevoroshkin. ***Hierarchical Search and the Discovery of Strategies.*** Technical Report, University of Michigan, 1993. Unpublished.
6. C bonell, Jaime G., et al. "An Overview of Machine Learning." In Michalski et al. (50).
7. Coad, Peter and Jill Nicola. ***Object-Oriented Programming.*** Englewood Cliffs, New Jersey: Yourdon Press, 1993.
8. Cooke, Joseph M., et al. ***NPSNET: Flight Simulation Dynamic Modeling Using Quaternions.*** Technical Report, Naval Postgraduate School, 1993. From Presence, Volume 1, Number 4, 1992.
9. Cormen, Thomas H., et al. ***Introduction to algorithms.*** Cambridge, MA/New York: The MIT Press/McGraw-Hill, 1990.
10. Davis, Lawrence, editor. ***Handbook of Genetic Algorithms.*** Van Nostrand Reinhold, 1991.
11. Davis, Lawrence, et al. "Temporary Memory for Examples Can Speed Learning in a Simple Adaptive System." ***From animals to animats 2: Proceedings of the Second International Conference on Simulation of Adaptive Behavior*** edited by Jean-Arcady A. Meyer, et al., MIT Press, 1993.
12. De Jong, Kenneth. "Genetic-Algorithm-based Learning." In Kodratoff and Michalski (40).
13. DeJong. ***An Analysis of the Behavior of a Class of Genetic Adaptive Systems.*** PhD dissertation, University of Michigan, 1975.
14. desJardins, Marie Ellen, editor. ***PAGODA: A Model for Autonomous Learning in Probabilistic Domains.*** Technical Report Number UCB/CSD 92/678, University of California, Berkeley, 1992.
15. Dorigo, Marco and Uwe Schnepf. "Genetics-Based machine Learning and Behavioral-Based Robotics: A New Synthesis." ***IEEE Transactions on Systems, Man, and Cybernetics, Vol. 23, No. 1***, 141–153. IEEE Press, 1993.
16. (Draft), Military Standard. ***Protocol Data Units for Distributed Interactive Simulation.*** Technical Report, Institute for Simulation and Training, and University of Central Florida, 1990. Contract Number N61339-89-C-0043.
17. Droddy, Vincent A. ***Multicriteria Mission Route Planning Using Parallelized A* Search.*** MS thesis, 93D-06, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1993.
18. Filho, J. L. Riberio and P. Treleaven. "Genetic Algorithm Programming Environments." ***Parallel Genetic Algorithms: Theory and Practice*** edited by Joachim Stender, 65–83, Washington, DC: IOS Press, 1993.
19. for Simulation, Institute and Training, "Standard for Information Technology Protocols for Distributed Interactive Simulation Applications, Version 2.0 (Second Draft)," 22 March 1993.

20. Forrest, Stephanie. *Parallelism and Programming in Classifier Systems*. Morgan Kaufmann, 1991.

21. Forrest, Stephanie, editor. *Proceedings of the Fifth International Conference on Genetic Algorithms*, Morgan Kaufmann, 1993.

22. Goldberg, David E. *Genetic Algorithms in Search, Optimization & Machine Learning*. Addison-Wesley, 1989.

23. Goldberg, David E., et al. "Genetic Algorithms, Noise, and Sising of Populations," *Complex Systems*, *6*:333–362 (1992).

24. Grant, T. J. "A review of multi-agent systems techniques with application to Columbus User Support Organisation." *Future Generation Computer Systems, Volume 7, Number 1* Elsevier Science Publishers B. V., 1992.

25. Grefenstette, John J., editor. *Proceedings of the First International Conference on Genetic Algorithms*, Lawrence Erlbaum Associates, 1988.

26. Grefenstette, John J. "A System for Learning Control Strategies with Genetic Algorithms." In Schaffer (72).

27. Grefenstette, John J. "Lamarckian Learning in Multi-agent Environments." *Proceedings of the Fourth International Conference on Genetic Algorithms*. Morgan Kaufmann, 1991.

28. Grefenstette, John J. "Lamarckian Learning in Multi-agent Environments." In Belew and Booker (3).

29. Gunsch, Major Gregg, et al., "On Applying Machine Learning to Develop Air Combat Simulation Agents," 1993.

30. Hammer, John M., "Verification and Validation of Knowledge Bases in Associate Systems," 1991.

31. Hipwell, Dean. *Pilot Decision Phases in Clips*. MS thesis, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1993.

32. Hluck, George. *Developing Realistic Behaviors in Adversarial Agents for Air Combat Simulation*. MS thesis, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1993.

33. Holland, John. *Adaption in Natural and Artificial Systems*. University of Michigan Press, 1975.

34. Holland, John. "Escaping Brittleness: The Possibilities of General-Purpose Learning Algorithms Applied to Parallel Rule-based Systems." *Machine Learning: An Artificial Intelligence Approach, Volume II* edited by R. Michalski, et al., Morgan Kaufmann Publishers, 1986.

35. Holland, John. *Adaption in Natural and Artificial Systems*. The MIT Press, 1992.

36. Holland, John, et al. *Induction: Processes of Inference, Learning, and Discovery*. The MIT Press, 1986.

37. Hoover, Stewart V. and Ronald F. Perry. *Simulation: A problem solving approach*. New York: Addison-Wesley Publishing Company, 1989.

38. Kargupta, Hillol, et al. *Ordering Genetic Algorithms and Deception*. Technical Report IlliGAL Report No. 92006, 117 Transportation Building, 104 South Mathews Avenue, Urbana, IL 61801: Illinois Genetic Algorithms Laboratory, Department of General Engineering, University of Illinois at Urbana-Champaign, April 1992

39. Kilpatrick, Freeman A. *An investigation of discovery-based learning in the route planning domain*. MS thesis, , School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1992.

40. Kodratoff, Yves and Ryszard S. Michalski, editors. *Machine Learning: An Artificial Intelligence Approach, Volume III*. Morgan Kaufmann Publishers, 1990.

41. Koza, John. "Hierarchical Genetic Algorithms Operating on Populations of Computer Programs." *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence, Volume I*. Morgan Kaufmann Publishers, 1989.

42. Koza, John. *Genetic Programming: On The Programming of Computers by Means of Natural Selection*. The MIT Press, 1992.

43. Laird, J. E., et al. "Soar: An Architecture for General Intelligence." In Rosenbloom et al. (68).

44. Laird, J. E. and P. S. Rosenbloom. "Integrating Execution, Planning, and Learning in Soar for External Environments." In Rosenbloom et al. (69).

45. Lewis, Ted G. and Hesham El-Rewini. *Introduction to Parallel Computing*. Prentice Hall, 1992.

46. Merkle, Laurence D. *Generalization and Parallelization of Messy Genetic Algorithms and Communication in Parallel Genetic Algorithms*. MS thesis, AFIT, 1992.

47. Meyer, J. A. and S. W. Wilson, editors. *From animals to animats: Proceedings of the first international conference on simulation of adaptive behavior*, MIT Press, 1991.

48. Meyer, Jean-Arcady and Agnès Guillot. "Simulation of Adaptive Behavior in Animats: Review and Prospect." In Meyer and Wilson (47).

49. Michalski, Ryszard S. "Understanding The Nature of Learning: Issues and Research Directions." *Machine Learning: An Artificial Intelligence Approach, Volume II* edited by Ryszard Michalski, et al., Morgan Kaufmann Publishers, 1986.

50. Michalski, Ryszard S., et al., editors. *Machine Learning: An Artificial Intelligence Approach*. Tioga Publishing Company, 1983.

51. Montana, D. and Lawrence Davis. "Training Feedforward Neural Networks Using Genetic Algorithms." *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence, Volume I*. Morgan Kaufmann Publishers, 1989.

52. Muruzabal, Jorge. *A machine learning approach to a problem in exploratory data analysis*. PhD dissertation, University of Minnesota, 1992 (9303384).

53. Olsan, James B. *Genetic Algorithms Applied to a Mission Routing Problem*. MS thesis, 93D-12, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1993.

54. Pearl, Judea. *Heuristics: Intelligent search strategies for computer problem solving*. Reading, MA: Addison-Wesley, 1985.

55. Press, William H., et al. *Numerical Recipes in C: The art of Scientific Computing*. New York: Cambridge University Press, 1988.

56. Ramsey, Connie L. and John J Grefenstette. "Case-Based Initialization of Genetic Algorithms." In Forrest (21).

57. Rich, Elaine and Kevin Knight. *Artificial Intelligence*. New York: McGraw-Hill, 1991.

58. Riolo, Rick L. "Bucket Brigade Performance: I. Long Sequences of Classifiers." *Proceedings of the Second International Conference on Genetic Algorithms* edited by John J. Grefenstette, Lawrence Erlbaum Associates, 1988.

59. Riolo, Rick L. *CFS-C: A Package of Domain Independent Subroutines for Implementing Classifier Systems in Arbitrary, User-Defined Environments*. University of Michigan, Ann Arbor: Logic of Computers Group, 1988.

60. Riolo, Rick L. *CFS-C/FSW1: An Implementation of the CFS-C Classifier System in a Domain that Involves Learning to Control a Markov Process.* University of Michigan, Ann Arbor: Logic of Computers Group, 1988.
61. Riolo, Rick L. *Let.* University of Michigan, Ann Arbor: Logic of Computers Group, 1988.
62. Riolo, Rick L. "The Emergence of Coupled Sequences of Classifiers." In Schaffer (72).
63. Riolo, Rick L. "The Emergence of Default Heirarchies in Learning Classifier Systems." In Schaffer (72).
64. Riolo, Rick L. "Lookahead Planning and Latent Learning in a Classifier System." In Meyer and Wilson (47).
65. Roitblat, H. L., et al. "Biomimetic Sonar Processing: From Dolphin Echolocation to Artificial Neural Networks." In Meyer and Wilson (47).
66. Rosenbloom, P. S. and J. Aasman. "Knowledge Level and Inductive Uses of Chunking." In Rosenbloom et al. (69).
67. Rosenbloom, Paul S., et al. "Introduction." In *The Soar Papers: Research on Integrated Intelligence, Volume One* (68).
68. Rosenbloom, Paul S., et al., editors. *The Soar Papers: Research on Integrated Intelligence, Volume One.* The MIT Press, 1993.
69. Rosenbloom, Paul S., et al., editors. *The Soar Papers: Research on Integrated Intelligence, Volume Two.* The MIT Press, 1993.
70. Russell, Stuart J. "Execution architectures and compilation." *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence, Volume 1.* 15–20. San Mateo, CA: Morgan Kaufmann Publishers, Inc., 1989.
71. Scales, D. J. "Efficent Matching Algorithms for the Soar.OPS5 Production System." In Rosenbloom et al. (68).
72. Schaffer, J. David, editor. *Proceedings of the Third International Conference on Genetic Algorithms*, Morgan Kaufmann, 1989.
73. Schapire, Robert E., editor. *The Design and Analysis of Efficient Learning Algorithms.* The MIT Press, 1992.
74. Simon, Herbert A. "Why should machines learn?." In Michalski et al. (50).
75. Smith, Stephen Frederick. *A Learning System based on Genetic Adaptive Algorithms.* PhD dissertation, University of Pittsburgh, 1980 (8112638).
76. Spears, William M. *Simple Subpopulation Schemes.* Technical Report, Navy Center for Applied Research in Artificial Intelligence, 1993. To be published.
77. Spears, William M. and Kenneth A. De Jong. "Using Genetic Algorithms For Supervised Concept Learning." *Proceedings of The 2nd International IEEE Conference on Tools for Artificial Intelligence.* IEEE Computer Society Press, 1990.
78. Spohn, Darren L. *Data Network Design.* New York: McGraw-Hill, 1993.
79. Wang, DeLiang and Michael A. Arbid. "Hierarchical Dishabituation of Visual Discrimination in Toads." In Meyer and Wilson (47).
80. Wilson, Stewart W. "Knowledge Growth in an Artificial Animal." In Grefenstette (25).
81. Wilson, Stewart W. "The Animat Path to AI." In Meyer and Wilson (47).
82. Wilson, Stewart W. and David E. Goldberg. "A Critical Review of Classifier Systems." In Schaffer (72).
83. Woolf, Henry B., editor. *Webster's New Collegiate Dictionary.* Springfield, Massachusetts: G. & C. Merriam Company, 1979.
84. Zhou, Hayong. "Classifier System with Long-term Memory in Machine Learning." In Grefenstette (25).

60. Riolo, Rick L. *CFS-C/FSW1: An Implementation of the CFS-C Classifier System in a Domain that Involves Learning to Control a Markov Process*. University of Michigan, Ann Arbor: Logic of Computers Group, 1988.
61. Riolo, Rick L. *Let*. University of Michigan, Ann Arbor: Logic of Computers Group, 1988.
62. Riolo, Rick L. "The Emergence of Coupled Sequences of Classifiers." In Schaffer (72).
63. Riolo, Rick L. "The Emergence of Default Heirarchies in Learning Classifier Systems." In Schaffer (72).
64. Riolo, Rick L. "Lookahead Planning and Latent Learning in a Classifier System." In Meyer and Wilson (47).
65. Roitblat, H. L., et al. "Biomimetic Sonar Processing: From Dolphin Echolocation to Artificial Neural Networks." In Meyer and Wilson (47).
66. Rosenbloom, P. S. and J. Aasman. "Knowledge Level and Inductive Uses of Chunking." In Rosenbloom et al. (69).
67. Rosenbloom, Paul S., et al. "Introduction." In *The Soar Papers: Research on Integrated Intelligence, Volume One* (68).
68. Rosenbloom, Paul S., et al., editors. *The Soar Papers: Research on Integrated Intelligence, Volume One*. The MIT Press, 1993.
69. Rosenbloom, Paul S., et al., editors. *The Soar Papers: Research on Integrated Intelligence, Volume Two*. The MIT Press, 1993.
70. Russell, Stuart J. "Execution architectures and compilation." *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence, Volume 1*. 15–20. San Mateo, CA: Morgan Kaufmann Publishers, Inc., 1989.
71. Scales, D. J. "Efficent Matching Algorithms for the Soar.OPS5 Production System." In Rosenbloom et al. (68).
72. Schaffer, J. David, editor. *Proceedings of the Third International Conference on Genetic Algorithms*, Morgan Kaufmann, 1989.
73. Schapire, Robert E., editor. *The Design and Analysis of Efficient Learning Algorithms*. The MIT Press, 1992.
74. Simon, Herbert A. "Why should machines learn?." In Michalski et al. (50).
75. Smith, Stephen Frederick. *A Learning System based on Genetic Adaptive Algorithms*. PhD dissertation, University of Pittsburgh, 1980 (8112638).
76. Spears, William M. *Simple Subpopulation Schemes*. Technical Report, Navy Center for Applied Research in Artificial Intelligence, 1993. To be published.
77. Spears, William M. and Kenneth A. De Jong. "Using Genetic Algorithms For Supervised Concept Learning." *Proceedings of The 2nd International IEEE Conference on Tools for Artificial Intelligence*. IEEE Computer Society Press, 1990.
78. Spohn, Darren L. *Data Network Design*. New York: McGraw-Hill, 1993.
79. Wang, DeLiang and Michael A. Arbid. "Hierarchical Dishabituation of Visual Discrimination in Toads." In Meyer and Wilson (47).
80. Wilson, Stewart W. "Knowledge Growth in an Artificial Animal." In Grefenstette (25).
81. Wilson, Stewart W. "The Animat Path to AI." In Meyer and Wilson (47).
82. Wilson, Stewart W. and David E. Goldberg. "A Critical Review of Classifier Systems." In Schaffer (72).
83. Woolf, Henry B., editor. *Webster's New Collegiate Dictionary*. Springfield, Massachusetts: G. & C. Merriam Company, 1979.
84. Zhou, Hayong. "Classifier System with Long-term Memory in Machine Learning." In Grefenstette (25).

Vita

Captain Edward O. Gordon was born 1961 in Trenton, New Jersey. After graduating from high school in 1979, he attended college at Lehigh University in Bethlehem, Pennsylvania, where he graduated in early 1984 after he earned his B.S.E.E. degree and received his commission through the R.O.T.C. program.

His first assignment was as officer in charge of the Gulf of Mexico Range telemetry reception and processing facility at Tyndall Air Force Base in Florida, where he provided the engineering guidance and direction to update the antiquated site to current technology.

In 1988, Captain Gordon was reassigned to Wright Patterson Air Force Base in Ohio, where he provided engineering support to the Aeronautical Systems Division. He worked as project engineer on two programs and headed the evaluation of computer hardware and software proposals during three source selections.

In 1992 he was accepted as a full-time graduate student at the Air Force Institute of Technology, where he is working to finish his Masters in Electrical Engineering. His next assignment is to the Defense Nuclear Agency in Alexandria, Virginia.

Permanent address: Jacobs Creek Rd
Titusville, NJ 08560

REPORT DOCUMENTATION PAGE

Form Approved
OMB No 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1993	Master's Thesis

4. TITLE AND SUBTITLE
Discovery Learning in Autonomous Agents Using Genetic Algorithms

5. FUNDING NUMBERS

6. AUTHOR(S)
Edward O. Gordon, Captain, USAF

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Institute of Technology, WPAFB OH 45433-6583

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/GCE/ENG/93D-10

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

As the new Distributed Interactive Simulation (DIS) draft standard evolves into a useful document and distributed simulations begin to emerge that implement parts of the standard, there is renewed interest in available methods to effectively control autonomous aircraft agents in such a simulated environment. This investigation examines the use of a genetics-based classifier system for agent control. These are robust learning systems that use the adaptive search mechanisms of genetic algorithms to guide the learning system in forming new concepts (decision rules) about its environment. By allowing the rule base to evolve, it adapts agent behavior to environmental changes. Addressed are the learning needs of autonomous aircraft agents, showing how multiple learning strategies are possible and that the best approach is a coherent combination of these. A design is described for a control system using a distributed filtering architecture and a genetics-based classifier system modified to support a phasing-rule niching system based on phase tags. Finally, a prototype system called the Phased Pilot Learning System (PPLS) is implemented based on this design and tested within a limited simulation environment. Results from empirical tests show that this approach is a viable alternative to other control methods.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Discovery Learning, Genetic Algorithms, Autonomous Agents	220
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL